PATENT ABSTRACTS OF JAPAN

(11)Publication number : 06-004498

(43)Date of publication of application : 14.01.1994

(51)Int.Cl. G06F 15/16
G06F 9/45

(21)Application number : 04-162882

(22)Date of filing : 22.06.1992

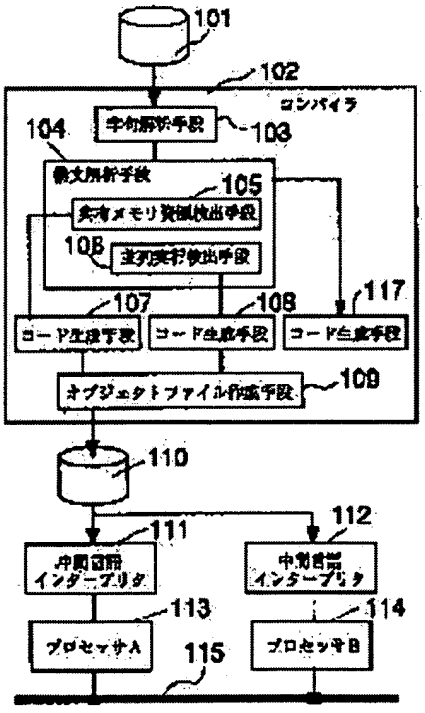
(71)Applicant : SEIKO EPSON CORP

(72)Inventor : NAGASAKA FUMIO

(54) MULTIPROCESSOR

(57)Abstract:

PURPOSE: To execute parallel processing without recompiling object codes even when the hardware constitution of a parallel processing system is changed by providing a mechanism for exclusively controlling access to variables shared at the time of parallel execution and dynamically performing the processing block assignment of the parallel execution realized by an interpreter for executing an intermediate language outputted by a compiler.



CONSTITUTION: This multiprocessor is provided with a compiling mechanism 102 having a means for analyzing program description for generating the access at least to shared memory source assignment and the means for analyzing the description for a program processing unit capable of parallel processing and performing processor assignment. Then, interpreting mechanisms 111 and 112 for executing the intermediate language outputted by the compiling mechanism 102 are provided to dynamically perform the processing block assignment of the parallel execution realized by the interpreting mechanisms 111 and 112.

LEGAL STATUS

[Date of request for examination] 21.06.1999

[Date of sending the examiner's decision of 18.12.2001

rejection]

[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]

[Date of final disposal for application]

[Patent number]

[Date of registration]

[Number of appeal against examiner's decision of rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

* NOTICES *

1.This document has been translated by computer. So the translation may not reflect the original precisely.
2.**** shows the word which can not be translated.
3.In the drawings, any words are not translated.

DETAILED DESCRIPTION

[Detailed Description of the Invention]
[0001]
[Industrial Application] This invention relates to the parallel execution processing by the multiprocessor from which architecture differs.
[0002]
[Description of the Prior Art] The system which performs homogeneous parallel processing by using two or more same processors is already known.
[0003] About the system which performs parallel processing by using two or more processors which have different architecture on the other hand, the approach of dropping on a direct absolute language by compile is learned. For example, the vector processor equipment of the large-sized computer stated by JP,62-34275,A etc. corresponds to this. However, by this approach, the difference of each architecture was not able to be absorbed and an execution environment equivalent to a homogeneous multiprocessor processor was not able to be realized.
[0004] Then, the method of making a pseudo code intervene between them is proposed by JP,63-41934,A, without compile dropping on a direct absolute language as an approach of solving the problem. By this approach, processing is performed by the compiler which generates intermediate-language object code, and the intermediate-language interpreter, and, thereby, the access frequency between CPU and external memory is decreased.
[0005]
[Problem(s) to be Solved by the Invention] However, conventionally [above-mentioned], since invention was not what realizes exclusive control of a share variable required for parallel processing, and distribution of a run unit, it had the trouble that parallel execution of the object program written by the juxtaposition description language was not carried out in fact.
[0006] The place which took the example in order that this invention might solve such a problem, and makes into the purpose is for a series of activities of recompiling to make [to change an object program, even when there are realizing parallel execution with the object program written in the system which performs parallel processing based on the juxtaposition description language specification by using two or more processors with different architecture, and modification of a processor unit, and] parallel execution there be nothing possible.
[0007]
[Means for Solving the Problem] In order to solve such a technical problem the multiprocessor processor of this invention A means to analyze programme description which generates access to shared memory resource assignment at least, and to direct exclusive control, The compile device in which it has a means to analyze description to the program manipulation unit in which parallel processing is possible, and to perform processor assignment, It is constituted by the interpreting device in which intermediate language which said compile device outputted is performed, and the device in which processing block assignment of the parallel execution realized according to said interpreting device is performed dynamically.

[0008]
[Example] Explanation of an example is performed according to each following item.
[0009] 0. Processing 1-3-1. argument list analysis processing which are the explanation 1-2. parallel execution of the compile 1-1. target system of 1. source code and a synchronization, the description approach 1-3. lexical analysis of exclusive control, and syntax analysis first (S404)
1-3-2. Extract of Dependency During Block (S405, S416)
1-3-3. share variable processing (S412)
2. Code Generation Means of Code Generation Means 1082-2. Shared Resource Access Procedure at Time of Code Generation 2-1. Parallel Execution by Intermediate Language (S418)
2-3. Specify Execution-Time Processing 3-5. Activation Processor of Execution-Time Processing 3-4. Shared Resource Access of Operating State 3-3. Juxtaposition Description Part of Communication Link 3-2. Intermediate-Language Interpreter between of Operation 3. Intermediate-Language Interpreter 3-1. Intermediate-Language Interpreters of Object File Creation Means 109. The supplementary 4-1. compiler of the procedure 4. explanation to give, Drawing 1 is the block diagram of a multiprocessor system suitable as one example of this invention, and its compiler at the beginning of mounting 0. of expansion 4-2-1. exclusive control of table 4-2. this example which an interpreter uses, and a shared resource management 4-2-2. intermediate-language interpreter. Compile of a source code 101 may be performed on a different computer from the target system. At this time, the intermediate-language object 110 is outputted by considering a source code 101 as an input. This intermediate-language object 110 is performed in the actual target system.
[0010] Drawing 2 is the block diagram of the personal computer 200 suitable as one example of the target system.
[0011] 1. The personal computer 200 which is the target system of explanation this example of the compile 1-1. target system of a source code controls user interface equipments, such as a display and an input unit, by the control unit 204, and receives actuation of a user with it. A processor 113 is a general-purpose microcomputer, performs processing of an operating system 203 and is performing management of a processor resource and a memory resource. The intermediate-language interpreter 111 is an independent process which calls the function of this operating system 203 and is performed. A process is a unit at the time of activation of the processor resource in an operating system, and memory resource assignment. A process consists of a field for saving the condition of the register of a current processor, and an object code field and a stack area by this example in a process header including the information for the identifier of a process and execution control, and memory management, and the case of interruption. When multiprocessing in a personal computer 200, multiple processes are started and processing is multiplexed by the schedule to a process. Moreover, the object code of the object program described with intermediate language starts the intermediate-language interpreter 111, respectively, and is performed on this interpreter.
[0012] A user can aim at improvement in functional by hardware addition to the processor 113 of a personal computer 200. A personal computer 200 has the system bus 202 for an external escape in order to respond to this purpose. The configuration of drawing 2 showed signs that the addition hardware 201 was extended using the system bus 202. Here, a processor 114 raises the processing speed of a personal computer 200, when the part of processing of a processor 113 is shared and carries out parallel execution by the approach of mentioning later. Such addition hardware may be called an accelerator from the role. A processor 114 executes the kernel program 205 in order to process transfer of the data through a system bus 202, interruption, etc. The intermediate-language interpreter 112 is a process managed by this kernel 205. In order that a processor 114 may perform these program executions independently in a processor 113, the partial memory 206 is used. Moreover, connection of a processor 114 and a system bus 202 is made by FIFO 207 and 208 which is the memory with sequence constituted so that it might put in and data could be taken out first the point. Address assignment is carried out and FIFO 207 and 208 is accessed by the room of a processor 113 from a processor 113.
[0013] 1-2. Description approach this example of parallel execution and a synchronization, and exclusive control describes clearly in a source code in the phase of programming language description of

the parallel execution by two or more processors. For this reason, the language specification which accepts juxtaposition description is required. Here, the language which added the predicate for juxtaposition description to Language Pascal is taken up and explained. In this example, the sentence started with start notations, such as a block, a call and assignment statement, if, while, repeat, and for, in the subprogram started in a procedure sentence or a function sentence in the syntax of Language Pascal for simplification of explanation is called a statement. Moreover, although the list of the "sentence" surrounded by begin...end usually calls it "compound statement (compound statement)", this is also called a statement here. The added specification is the following two kinds.

[0014] (1) cobegin, coendcobegin, and coend are the initial statements and termination sentences of a block which accept parallel execution, respectively.

[0015] (2) A monitor (monitor) mold monitor (monitor) mold is mold attachment (typing) which declares the so-called share variable accessed by two or more blocks by which parallel execution is carried out. Although it can also consider that a monitor mold is the format of abstracting and expressing a shared resource, a program top gives the resource which can be accessed only in the specified procedure. The method of dealing with the exclusive control and the synchronization to a share variable by this approach is detailed for two or more well-known examples (for example, Kazunori Ueda work: juxtaposition programming language, information processing, Vol.27, No.9, pp.995-1004 (1986), Brinch Hansen, :The Programming Language Concurrent Pascal written by P., IEEE Trans.Software Eng., Vol.1, No.2, pp.199-207 (1975), etc.).

[0016] Drawing 3 is the explanatory view of the program which used the monitor. Procedure 305 generates the data of a certain integer type, and a program 300 is the example which described the contents of processing that procedure 306 consumed it one after another, by the juxtaposition description language. Procedure 305 is the processing which receives data from an external device, and this corresponds, when procedure 306 processes and displays the data. By having declared Variable buf here as buffertype which is a monitor mold, the variable (a variable name is buffer) 308 which is the actual condition of the shared resource on memory cannot be accessed in the procedure 302 or 303 which the monitor type declaration 301 interior defined. Moreover, each variable in a monitor is initialized in procedure 304. If the producer procedure 305 tends to perform substitution to a variable 308 by procedure buf.append (v), processing 302 will be performed in fact. Supposing all the contents of buffer which is already an array variable 308 are filled at this time, this demand will make queue putq and will be kept waiting (this was displayed as procedure wait (putq)). (309) When other, after addition of the element to an array variable 308 is performed and a pointer is updated, procedure signal (getq) is called (310). If procedure signal (getq) has the processing call of the data acquisition which makes queue getq and is kept waiting, it will perform it, and it performs processing which returns a result. If the consumer procedure 306 tends to read one element of the contents of the variable 308 by procedure buf.fetch (v), processing procedure 303 will be performed. If the contents of buffer which is an array variable 308 are empty at this time, this demand will make queue getq and will be kept waiting (procedure wait of processing 311 (getq)). When other, after the contents of the array variable 308 are read and renewal of a pointer is performed, procedure signal (putq) is called (312). If procedure signal (putq) has a procedure call in queue putq like the above, it will perform it and will return a result. It is considered that the structure of a monitor mold is a resource under management of the block with which the data of a monitor mold were declared. Access to this resource is performed by the intermediate-language interpreter in this example. Since the processing of an interpreter itself is successive processing, it can guarantee the exclusive control to a share variable, and a processing synchronization as a result. The intermediate-language generating approach at the time of detecting a monitor type definition is described below.

[0017] 1-3. The compiler 102 which read the processing source code 101 of lexical analysis and syntax analysis performs lexical analysis with the lexical-analysis means 103. Here, ejection of the reserved word of a operator, a special kind notation, and programming language is performed to identifiers, such as procedure, a variable, a constant, a Hollerith constant, a data type definition, and a character string, a value, and a pan by reading the character string which appears in a source code 101 one by one. This

reading result is recorded in the working area of a compiler 102.
[0018] Next, the syntax-analysis means 104 is carried out by considering the data of the reading result of the lexical-analysis means 103 as an input. since the syntax-analysis mark of this example is the syntax of the property in which the syntax of Language Pascal is called LL (1), it performs well-known downward analysis (the well-known example about downward analysis -- : written by Ikuo Nakada -- there are a large number, such as a compiler and Sangyo Tosho Publishing (1981)). In language with the block structures, such as Language Pascal, a variable name and a block name make the inside of the declared block a scope. For this reason, the information which expresses the appearance sequence of a block as the depth of a block (when the depth of a block is the same) is required for the decision of the value of a variable, and reference. With the syntax-analysis means 104, it considers that the level of program initiation is block level =0, and processing is advanced. Then, whenever one step of condition of the nest of a block becomes deep, the value of block level is carried out +one. Similarly, whenever it slips out of one step of nested structure, the value of block level is carried out -one. That is, a main program is block level =0 and the procedure declared within a main program is block level =1.
[0019] The part into which the syntax-analysis means 104 of this example differs from the conventional approach is the point of processing the shared memory resource detection means 105 and the parallel execution detection means 106 in the interior. The former defines the variable accessed by two or more batches by which parallel execution is carried out, and generates the activation code which establishes the procedure to the variable. The latter generates the code for the procedure performed by juxtaposition and a function call.
[0020] The procedure of the syntax-analysis means 104 is explained using the flow chart of drawing 4 . Since syntax of language with the block structure is analyzed, the syntax-analysis means 104 of this example performs recursive downward analysis. That is, if syntax analysis within block detection and a block is performed and the definition of a block is in a block further, its processing will be called recursively and block detection will be performed. This can be written as follows with false programming language.
[0021]
procedure block (....); var ident : Predicate; ...; begin (* taking out the following predicate ident substitution *) ident : = [next_word_in_source;] if (ident ='procedure') or (ident ='function') then block (...) (* recursive *) else begin .... Other functors ...; end end; If procedure or declaration of a function is detected and it goes into a block in order to control this repeat correctly (S401), block level will be carried out +one (S402), and a block number will be carried out +one (S403). On the other hand, when termination of a block is detected, block level is carried out -one (S423), and it investigates whether as a result, the magnitude of block level is smaller than 0 (S424). When block level becomes smaller than 0, it is the level of a main program, and it means detecting termination of a block and it is shown that syntax analysis of a program was completed. Moreover, with [block level] zero [or more], further, functor may continue and continues processing. The above is the outline of the flow of processing of the syntax-analysis means 104. next, each **** -- it ** and also explains just.
[0022] It is necessary to record information, such as stack area size used for the starting address of an activation code, the storing location of a variable table, and argument delivery about each processing block, in processing of syntax analysis. This example uses the execution-time block managed table 501 and the block number managed table 801 for this purpose. Drawing 5 is drawing explaining the DS of the execution-time block managed table 501 used by this example. The execution-time block managed table 501 takes the format of the array which uses DS 502 as one element. DS 502 is constituted by a block number 503, the chain 504 to a low order block, the offset 505 on object code, the depth 506 of the stack consumed to argument delivery, and the chaining address 507 to an argument list.
[0023] Explanation of return procedure is again continued to a flow chart.
[0024] By processing S403, a block number with all unique processing blocks in a source code (procedure, function) is given. In order to hold the block number of each processing block, and the relation of block level, in processing of this example, the block number managed table 801 shown in drawing 8 a is generated. One element of a block number managed table is DS which consists of a block

number 802, block level 803, and a table entry 804. In processing S403, record of the value of a block number 802 and the block level 803 is performed. Moreover, processing S403 is what byte in location in the object code starting position of all programs to a relative position, or the starting position of the code generation of a current processing block calculates it. This value is recorded on DS 505 of an execution-time block managed table.

[0025] Each block (procedure or function) can receive an argument at the time of the call. At this time, by the intermediate-language interpreter, the memory secured as a stack area is loaded with that argument, and a processing call is performed. So, in order to carry out the schedule of the procedure which exists at the time of parallel execution to another processor and to perform it, it is necessary to interpret the argument arranged in this stack area, and to copy to the working area of the intermediate-language interpreter of another processor. For this reason, argument list analysis processing (S404) is performed. This processing records an argument list on DS 507 of the execution-time block managed table 501 as a connection list. The following knot explains the detail of this processing.

[0026] Then, link generating during a block is performed (S405). This generates information required for the processing block transfer at the time of parallel execution. It mentions later for details.

[0027] This after treatment registers it into an identifier table, when constant declaration is detected (S406) (S407), and when a type declaration is detected (S408), it performs registration to a type-declaration dictionary (S409). Usually, in syntax-analysis processing, no identifiers other than the reserved word which appears in a source program distinguish a constant identifier, a variable name, a procedure identifier, a function name, and a variable-type name, but register them into an identifier table, and take the approach of recording that object type on this front Naka with an identifier in many cases. This is an approach rational when detecting beforehand un-arranging [of the overloading (a function name and a constant identifier are identitas etc.) of an identifier]. This example also depended management of the identifier (identifier) in the syntax-analysis means 104 on this approach.

[0028] However, when variable declaration is detected (S410) and a monitor mold variable is detected in the shared memory resource detection means 105 (S411), share variable processing (S412) which is later mentioned besides the registration to the usual identifier table is carried out. If it is variables other than a monitor mold, tabulation about the variable within a block will be performed (S413), and the code generation of local parameter assignment will be processed further (S414). Here, the starting address on the working area of the compiler 102 of the variable table created about the variable within a block by processing S413 is recorded as a value of the table entry 804 in the block number managed table 801.

[0029] Next, the flow of the processing about functors other than block initiation is explained.

[0030] in this example, since juxtaposition description is accepted in language specification, it is being required that the part which should be carried out parallel execution should be boiled clearly, and should be described in a source code. As for the processing block which is not included in the range surrounded by cobegin and coend, in other words, the activation code in a serial-processing environment is generated. If this is seen from the position of the syntax-analysis means 104 and the code generation means 107, 108, and 117, the activation code of the block of those other than the batch which had parallel execution specified by cobegin and coend will perform the same code generation as the time of the usual compile processing. Then, the syntax-analysis means 104 carries out the parallel execution detection means 106. The processing is processing made into parallel execution flag = [a false] (S421), when reserved word cobegin was detected in the functor of a source code, it sets up with parallel execution flag = [truth] (S420) and reserved word coend is detected. Generation of an actual activation code (intermediate-language sentence) is performed judging the truth of this flag.

[0031] It can be judged that the taken-out predicate is an identifier, and it is the processing which calls other processing blocks supposing it is a procedure identifier or a function name, as a result of searching identifier table registration moreover (S415). When this decision is [truth], link processing during a block is performed first (S416), and then, code generation of a block call is performed (S425). At this time, the contents of the "parallel execution flag" mentioned above are inspected, and when it is flag = [truth], the code generation means 108 is performed. In the case of others, the code generation means 117 is performed.

[0032] Moreover, when the taken-out predicate has the structure of the identifier "monitor mold variable name" +"."+ "a procedure identifier", an identifier table is searched and an identifier exists by the monitor mold variable name, it can be judged that it is a monitor mold procedure call (S417). In this case, shared resource access procedure generation processing (S418) is performed. The code generation means 107 is further processed as a substructure of this processing, and it acts as Shigeo of the intermediate-language sentence.
[0033] The above is the outline of processing of the syntax-analysis means 104. Next, sequential explanation is given about actuation of each part.
[0034] 1-3-1. Argument List Analysis Processing (S404)
As everyone knows, in the language of a Pascal system, when an argument is passed to procedure, there are two kinds in the format of refer to the argument. They are "the argument (call by value) which passes a value", and "the argument (call by reference) which passes an identifier." As internal representation of an argument reference mold, in the case of value delivery, "reference mold =0" and when an identifier was passed, it was referred to as "reference mold =1" in this example.
[0035] An argument list is "(it is started by "and written by the pair of the data type (an integer, an array, alphabetic character, etc.) of a variable name and a variable.) which follows a block name in functor. Moreover, termination of an argument list is ".; It is ". Furthermore, "the argument which passes an identifier" is started by reserved word "var". When reserved word "var" is detected, argument list analysis processing S404 is set to reference mold =1, and when other, it is set to reference mold =0. Next, the data type of a variable is analyzed and the value of the internal representation given to the data type is taken out. The value (integral value) unique as internal representation is altogether given to usable data type. The analysis processing S404 adds the value of this "reference mold and variable type" to the list of DS 509 which continues from the chaining address 507 of an execution-time block managed table. DS 509 takes the data type of a connection list, and the tail of data is the chaining address. The chain 509 from the chaining address 507 is referred to as an argument list at the time of code generation processing.
[0036] 1-3-2. Extract of Dependency During Block (S405, S416)
In the processor of this example, the target block by which parallel execution is carried out will be transmitted to the partial room of another processor, and will be performed (if it is possible). When for that a certain processing block has been arranged to other processors, the procedure called from this processing block and all the functions are taken out, and to arrange this copy in the partial memory of the same processor as the aforementioned processing block is desired. This is processing for removing the need of carrying out interprocess communication whenever a certain processing block calls a low order block. When it takes notice of a certain processing block set as the object of parallel execution, the management tool which can specify the block which that processing block calls (it is written below that this is subordinate) should just be carried out by this implementation. This example realized this management tool by holding the DS holding the dependency during each processing block on the execution-time block managed table 501.
[0037] The processings S405 and S416 in the flow chart of drawing 4 are processings which describe the link during an actual block in this execution-time block managed table.
[0038] When a program is described like the source code of drawing 6 from the property of the language of this example, if the block number given by processing S403 at the time of the block structure between batches and syntax analysis is illustrated, it can be shown like drawing 7 . Although it is accessible to the variable of high order structures, such as procedure p32 and p3, inside processing of the procedure p321 which is a block number 11, for example if the view of the scope (scope) given to identifiers, such as a variable name and a procedure identifier, is followed, access to the variable of procedure p31 and p2 grade is not allowed.
[0039] In order to take out the hierarchical relationship of a processing block, maintaining the relation of such a variable scope correctly, processing S405 processes referring to the block number managed table 801. This processing was shown in the flow chart of drawing 9 . If the block level of a self-block is one of 0 and 1, since the self-block is declared by the top level and it is not subordinate to a high order

hierarchy, link generating is unnecessary (S901). In other than this, the element in a table is gone back from the location of a self-block of the block number managed table 801, and the element in which the value of block level has a value smaller [one] than the value of the block level of a self-block is looked for (S902). The block number of the element is taken out (S903), the location of the block with the block number 503 which is in agreement out of the execution-time block managed table 501 by using the value of the block number as a key is found, and it accesses to the element. The chain 508 from the chaining address 504 to the low order block of this element is generated (S904). The chain to a self-block is recordable about the block located in high order structure by the above processing. Drawing 8 b is drawing which took the program of drawing 6 for the example and explained the link motion during a block. In order to make it intelligible, the arrow head showed the situation of the link during a block to the right-hand side of the contents of the block number managed table 801. A block number when a block number is given according to the block structure like drawing 7 from the program of drawing 6 = the procedure p32 of 10 is subordinate to the procedure p3 of block number =5. As shown in drawing 8 b, since the value of the block level of procedure p32 is 2, if the block number managed table 801 is gone back in search of an element which is block level =1, it can detect the location of the procedure p3 of block number =5.

[0040] On the other hand, processing S416 is processing which generates the chain to the block which a self-block calls. This processing is more simple. That is, if the identifier is procedure or an identifier of a function as a result of searching an identifier table according to the identifier in functor, the block number will be taken out. The value of this block number is only added to the chain 508 following the chaining address 504 of the location of a self-block of an execution-time block managed table. Moreover, if the element of the same block number is already on a chain in generation of this chain 508, in order to avoid duplication, an addition for a chain is not performed.

[0041] 1-3-3. share variable processing (S412)

The share variable processing S412 secures the field of the activation code accompanied by access to an actual monitor mold variable. Next, the object code of the monitor mold procedure prepared for this field as standard procedure of an intermediate-language interpreter is loaded. At this time, a unique number is given to each monitor mold variable at the order declared within the source code. This is recorded on the shared resource managed table 1001 with the DS shown in drawing 10 . One element of the shared resource managed table 1001 consists of values 1005 of offset to the resource number 1002, the share variable name 1003, the block number 1004 of the processing block which declared the shared resource, and the object code 1006 of monitor mold procedure.

[0042] 2. Explain the code generation by intermediate language, next the generation procedure of intermediate language. Since explanation was easy, a part of table which explains the function about intermediate language was illustrated to drawing 11 . However, TOP expresses the address of the maximum upper case of the stack mold data area taken to the working area of an intermediate-language interpreter, and SP expresses the pointer indicating a stack mold data area. The interpreter stack of this example is the memory structure of the FILO mold which updates a pointer in the direction where an address value is big, when newly accumulating an element on the one where an address value is smaller and taking out an element. Moreover, the address with which Identifier x is stored is expressed in writing with addr (x).

[0043] An intermediate-language interpreter takes out the value in which the instruction written to be LODV addr (x) is stored to Address addr (x), and interprets it as instruction "store it in the maximum upper case of the stack at the time of activation of an intermediate-language interpreter." (In the following explanation, since description is easy, this is only expressed in writing with " LODV x ".) An instruction called LODA addr (x) evaluates the value of the address of addr (x) itself, and puts it on the maximum upper case of a stack again.

[0044] LODN obj is the shared resource of a processing block or a monitor mold, and obj takes out the value of the block number of an object, or a resource number, and puts it on the maximum upper case of a stack.

[0045] LOD n takes out the data which calculate the address with a depth of +n bytes from a stack

pointer current in the inside of the stack which an interpreter uses, and are in the address, and copies them on the maximum upper case of a stack.
[0046] ALOC n n Carries forward a current stack pointer toward a direction with few addresses (getting it blocked and subtracting only n), and assigns the field for a local parameter.
[0047] UALC n is the instruction which makes ALOC n and a pair, adds n to the value of a stack pointer, and is ALOC n. The field vacated with an instruction is returned.
[0048] CALL addr (x) returns to the instruction location of a degree which performed CALL X previously, when processing branching is carried out to the head address of Identifier X and instruction word RET is detected.
[0049] ADD applies the value of a stack maximum upper case, and the value in the next location of a stack, carries forward a stack pointer in the direction whose one-step stack decreases, and stores an addition result in the new maximum upper case of a stack.
[0050] In addition to this, intermediate language has instruction word, such as data processing and conditional-branching processing. Neither of the instructions used the register of a processor as an accumulator, but was made into the format of performing math operation on a stack. Needless to say, as long as the specification of this intermediate language is unified between the interpreters of the intermediate language which carries out parallel execution, is clear and is, they may be other specifications. Moreover, the stack used here differs from a processor stack in the linear room taken to the working area of an intermediate-language interpreter.
[0051] Drawing 13 is the explanatory view which described a part of processing of intermediate-language description generation with false programming language. Description shows the code generation procedure 1301 at the time of a block call, and the code generation procedure 1303 at the time of detecting cobegin and a coend sentence. The code generation procedure 1302 to an argument is included in processing at the time of a block call. In addition, although it needed to be intermediate-language generated in the case of arithmetic operation, since it was the same as the conventional approach, it omitted. Processing of intermediate-language generation takes the approach of carrying out intermediate-language generating according to the Ruhr applied to the condition, when some conditions are detected. Therefore, processing is actuation which repeats processing incorporating detection of conditions, and evaluation of the identifier taken out by syntax analysis to the Ruhr-ized code generation, a value, an address, etc.
[0052] It explains how an example is given to below and the above-mentioned intermediate language is generated in an activation code.
[0053] 2-1. Explain generation of description by code generation means 108 intermediate language at the time of parallel execution using drawing 12 . When the definition of the procedure p1 shown in a source code 1200 appears, the syntax-analysis means 104 performs argument list analysis to the appearance shown in the flow chart of drawing 4 (S404). The 1st argument 1201 starts in Identifier x. The syntax-analysis means 104 judges this to be "the argument which passes a value." Consequently, a reference mold = 0 is recorded, and since the mold of an argument is next integer, 1 which is the internal representation of integer type is recorded as a "model name." The 2nd argument 1202 starts in reserved word var. Therefore, the syntax-analysis means 104 judges this to be "the argument which passes an identifier", and "reference mold =1" and "model name =1" are recorded. Consequently, an argument list is formed of the chain 509 from the chaining address 507 of the execution-time block managed table 501. If the contents of the argument list about procedure p1 are taken out, it is illustratable like 1203.
[0054] When description 1204 appears in a source code, when parallel execution is not carried out, the code generation means 117 is carried out, and the intermediate-language description 1205 is generated according to the contents of the argument list 1203. Generation of this description is what was depended on the approach of arranging the instruction word Ruhr-ized by the order which appears in an argument list 1203, and is the well-known approach.
[0055] In this example, different intermediate-language description from the conventional approach is generated about the part which carries out parallel execution. By this example, the procedure and the function with which parallel execution is allowed do not accept any variable parameters other than a

monitor mold variable as a prerequisite. This is natural if a variable parameter considers that it is the argument which accepts substitution. It considers as the procedure which has procedure p2 in the 1st argument, and has the monitor mold variable of identifier delivery in the integer of value delivery, and the 2nd argument, and procedure p3 is considered as procedure without an argument. If description 1206 appears in a source code, the syntax-analysis means 104 of this example will perform processing of the following procedure.
[0056] Procedure 1 The parallel execution detection means 106 detects a cobegin sentence, and makes a parallel execution flag [truth] (the flow chart of drawing 4 processing S420).
[0057] Procedure 2 Description"p2(a, b);" is analyzed in the syntax-analysis means 104, and it is detected that it is a block call (S415).
[0058] Procedure 3 The contents of the parallel execution flag are inspected. Here, since it is [truth], the code generation means 108 is performed.
[0059] Procedure 4 By the code generation means 108, the identifier table of the working area of a compiler 102 is searched, and the block number and argument list of procedure p2 are specified.
[0060] Procedure 5 From the contents of the argument list, it can know that the 1st argument is value delivery (reference mold = 0), and it is integer type.
[0061] Procedure 6 The code generation means 108 outputs the predicate of intermediate language according to the generation Ruhr. The Ruhr here is "the argument of value delivery outputs an instruction word {LODV x} format."
[0062] Procedure 7 From the contents of the argument list, the 2nd argument can know that it is a monitor mold with the variable of identifier delivery.
[0063] Procedure 8 "A monitor mold variable parameter outputs an instruction word {LODN obj} format" is used as the Ruhr, and intermediate-language description is outputted.
[0064] Procedure 9 The intermediate-language description which calls procedure p2 is generated. here -- the Ruhr at the time of parallel execution -- "a procedure call outputs an instruction word {LODN obj, CALL addr (coexec)} format" is applied.
[0065] Hereafter, processing is similarly performed about procedure p3. Furthermore, the following procedure is processed.
[0066] Procedure 10 The parallel execution detection means 106 detects a cobend sentence, and makes a parallel execution flag a [false] (the flow chart of drawing 4 processing S421).
[0067] Procedure 11 According to the Ruhr at the time of parallel execution termination, instruction word {CALL addr (sync)} is outputted.
[0068] Description 1207 is outputted as a result of the above procedure. Procedure coexec and sync are the inclusion procedure of an intermediate-language interpreter here. The knot of "execution-time processing of a 3-3. juxtaposition description part" explains actuation of these inclusion procedure.
[0069] In the code generation means (S418) syntax-analysis means 104 of 2-2. shared resource access procedure, when the functor accompanied by access to a shared resource is detected, processing S418 is performed. Generating of the intermediate language accompanied by access to a shared resource can be shown like 1304 of drawing 13 , if it expresses in writing with false programming language. In order to perform access to a monitor mold variable, it may be specified the case where a monitor mold variable is specified in procedure as a non-local parameter, and in an argument list.
[0070] The former code generation is simple. If the case where a code is performed is shown using drawing 14 a, an intermediate-language interpreter will place the argument 1404 to monitor mold procedure on a stack, will put the value 1405 of a monitor mold resource number on an upper case further, and will perform the procedure call of append and fetch. If this is seen from a code generation side, it is realizable by performing intermediate-language generating by the following procedure.
[0071] Procedure 1 The processing S418 of the syntax-analysis means 104 searches the shared resource managed table 1001 by using as a key the identifier of the monitor mold described in functor, and takes out a resource number. Procedure 2 The code generation means 107 is called.
[0072] Procedure 3 The procedure described by the program 1304 is performed. The argument to monitor mold procedure is put on the stack maximum upper case 1404 by object code generation of a

just before. Instruction word {LODN obj} is outputted following this.
[0073] Procedure 4 Only append call appearance judges only fetch call appearance, and an syntax-analysis result outputs instruction word {CALL addr (append or fetch)} for it.
[0074] On the other hand, there are two or more shared resources and monitor type parameter is needed under demand of wanting to distinguish the resource accessed though the same procedure is used by the case. The code generation in this case is a little complicated. The code containing monitor type parameter is the case where description 1208 appears in a source code etc. It explains using the hysteresis 1401 of use of the stack at the time of activation of drawing 14 b.
[0075] The argument at the time of a procedure call is accumulated on a stack. Then, if procedure is actually called, in order to secure the local variable within procedure, an instruction {ALOC N} is executed, and the partial field 1402 is secured. Furthermore object codes, such as an operation, consume a stack and a current processing result is arranged on the maximum upper case 1404 of the stack in this time. This is an argument to monitor mold procedure. Next, the resource number of a monitor mold is put on the maximum upper case of a stack. For this reason, instruction word {LOD+d} is performed and a value is copied at the stack top like an arrow head 1403 from the location of the depth which is +d bytes of a stack. Then, the procedure call of append and fetch is performed. If this is seen from a code generation side, it is realizable by performing intermediate-language generating by the following procedure.
[0076] Procedure 1 The field for a local parameter is developed. For this reason, instruction word {ALOC N} is outputted.
[0077] procedure 2 others -- object code generation is carried out.
[0078] Procedure 3 If it detects by monitor mold procedure call, the monitor mold variable in an argument list will calculate the depth to the location put on the stack. Instruction word {LOD+d} is generated using a count result as d.
[0079] Procedure 4 Only append call appearance judges only fetch call appearance, and an syntax-analysis result outputs instruction word {CALL addr (append or fetch)} for it.
[0080] Procedure 5 If , object code generation of further others is carried out. It outputs in order of instruction word {UALC N, RET} after that.
[0081] The object code 1209 by intermediate language is outputted from the description 1208 of a source code as a result of the above procedure.
[0082] 2-3. Object code description is obtained by the procedure in which the object file creation means 109 carried out **** of operation. An object file generation means outputs the intermediate-language object 110 by considering as an input object code description generated by the code generation means 107, 108, and 117. The block number with which the procedure call was connected to the identifier (identifier) of procedure has described object code description. Since this was substantially equivalent to the identifier, the above-mentioned explanation explained it using the procedure identifier. However, it is on the location of what byte the thing which is the need is recorded by the relative value from the head location not in the identifier of procedure but in the program of the procedure at the time of activation. This information is held according to DS 505 of the already explained execution-time block managed table 501. Then, if the object file creation means 109 reads object code description and the call of a block name is detected, it will search the execution-time block managed table 501 by using this block name (correctly block number) as a key, and will take out the relative position 505 in an activation code. Next, the block name of intermediate-language description is transposed to the value of a relative position 505 with this value.
[0083] The monitor mold variable / procedure which is a shared resource similarly are specified by the resource number by object code description. The object file creation means 109 searches the shared resource managed table 1001 by using a resource number as a key also about this, takes out the value 1005 of the offset on an activation code, and replaces the part described as a resource number in object code description.
[0084] Following the above processing, the object file creation means 109 is added after object code describing the contents of the execution-time block managed table 501 and the shared resource managed

table 1001, and it carries out a file output as an intermediate-language object 110.
[0085] 3. The compiler processor which carried out communication link **** between intermediate-language interpreter 3-1. intermediate-language interpreters does not need to be performed on the target system. It is possible to generate the intermediate-language object 110 by using other computer systems as host equipment. A file output is carried out and the generated intermediate-language object is transmitted to the target system by a magnetic-disk medium, Semi-conductor ROM, means of communications, etc. Or it is also possible to perform the compiler of the direct above-mentioned configuration with the target system, and to generate the intermediate-language object 110 from a source code.
[0086] An intermediate-language interpreter is a processor which operates considering this intermediate-language object 110 as an input. It may be a requirement that given intermediate language can be interpreted and performed, for this reason what kind of implementation gestalt is sufficient as an intermediate-language interpreter. But a simple example is an approach using the equipment which continues an endless loop as shown in the flow chart of drawing 16 . This example started the process 111 on the operating system 203 which a processor 113 performs, as drawing 2 showed, and it realized processing of the flow chart of drawing 16 according to this process 111. Moreover, in the processor 114, the process 112 was started on the kernel 205 which supports interruption processing etc., and flow of the same processing was realized. Therefore, processes 111 and 112 are called an intermediate-language interpreter (it omits Following IPR) here. Moreover, the processor to which starting of a main program was performed for explanation is called a local processor. Moreover, after detecting parallel execution description, the processor which a part of processing blocks are distributed and bears parallel execution is called a remote processor. The intermediate-language interpreter performed on Master IPR, a call, and a remote processor in the intermediate-language interpreter on the computer by which the startup of a main program was performed similarly is called SUREBU IPR.
[0087] The process which operates as IPR takes the configuration expressed as a process 1702 of drawing 17 . A process 1702 consists of the process header on which the information for process control is recorded, the activation coding region to which the processor program counter 1703 points, a working area, and a processor stack 1711 to which the processor stack pointer 1710 points. The object code 1704 of an interpreter program and the object code 1705 of an inclusion procedure group are stored in an activation coding region. The coding region 1706 by which intermediate-language object code is arranged, and the stack area 1707 for intermediate language are arranged in the working area of a process. Moreover, the instruction pointer 1708 for intermediate language and the stack pointer 1709 for intermediate language are put on the same working area, in addition the execution-time block managed table 501, the shared resource managed table 1001, and the execution-time processor managed table 1701 are placed. Using FIFO 207 and 208, IPR 111 and 112 can communicate and can suit. If FIFO208 has data writing, an interrupt will occur in a processor 113 through a system bus 202. This interruption is taken out by the manipulation routine of an operating system 203, and is transmitted to IPR111. Moreover, address assignment in the address space of a processor 113 is performed to FIFO207 by the system bus 202. For this reason, if a processor 113 performs data writing to this address, FIFO207 can generate the interruput signal to a processor 114. By the interrupt service routine, the kernel 205 of a processor 114 takes out the data written in FIFO207, and transmits them to IPR112.
[0088] It can be considered that IPR 111 and 112 is the process which communicates and suits using this FIFO 207 and 208. Two or more hardware added to a personal computer 200 is permissible on the design of this processor. For this reason, the communication link of Master IPR (in this case, IPR111) and SUREBU IPR is not limited to the communication link of 1 to 1. So, in this example, the data stream which has the DS of the format of 1501, 1505, or 1510 shown in drawing 15 in the communication link between the processes which operate as IPR is used. In this DS, the processor number 1503 is a number of each processor determined as a meaning by the system, and a process number 1504 is a number which each processor assigned at the process to the process generate time. Therefore, even if multiple processes exist, the process which is specifying both processor number 1503 and process number 1504 can be specified. On the other hand, being used as a command 1502 is six, a

{connection lock}, {loading}, {activation}, {a release}, a {fetch}, and {appending}. IPR which received communication links 1501 and 1505 returns the message which consists of DS of 1510 as a response. Five, a {busy}, a {ready}, {acceptance}, {Dawn}, and {a fail}, are used as a condition code 1511 here.
[0089] According to the flow chart of drawing 16 , actuation of IPR is explained below. If a command is received (S1601), the contents will judge IPR in a {connection lock} (S1602). If it is a {connection lock}, the condition code inside IPR will be made into a {busy} (S1603), and connection processing S1604 will be performed. The connection processing S1604 takes out the processor number 1503 and a process number 1504 from the data stream which received, and records them on an internal working area, and a condition code {acceptance} is returned to the process which generated {connection lock} processing. After this, after receiving a command, it judges whether it is the process from which the process which transmitted this was set as the object of connection processing (S1605), and if it is No, a condition code {a busy} will be returned (S1606). When the command from the process set as the object of connection processing is received, one command of {loading}, {a release}, and {activation} is executed. {Appending} and a {fetch} command are answered also to processes other than the connected process.
[0090] When a command is {a release}, a condition code is made into a {ready} (S1607), and the processor number and process number which were recorded for connection processing are discarded, and it returns to the command reading processing S1601.
[0091] If a command is {loading}, load processing S1608 will be performed. The load processing S1608 reads into the working area of SUREBU IPR the object code 1507 by which intermediate-language description was carried out, then reads the stack initialization data 1508, stores the data of an initial state in a stack according to directions of this data, and sets up a stack pointer. Then, the table initialization data 1509 are read and the contents of the execution-time block managed table 501 and the shared resource managed table 1001 are initialized.
[0092] When a command is {activation}, it goes into a running state and goes into executive operation (S1609) serially from the instruction which the instruction pointer of a coding region 1706 specifies. Serially, executive operation S1609 performs the instruction word processing group 1610, and also is incorporated according to the contents of processing, and performs the contents of processing of the procedure processing group 1611. Moreover, when it is in the processing loop formation of activation serially, polling processing S1612 is performed after an instruction execution. This is performed from the need of inspecting the queue from a communication link port, respectively, when checking processing termination of SUREBU IPR after sync instruction word activation, and when carrying out a {busy} response from other IPR(s) at a demand.
[0093] The above is the outline of IPR of operation.
[0094] 3-2. There is operating state of initial starting in IPR of operating state this example of an intermediate-language interpreter. This operating state is operating state made by the OPERETIGU system 203 immediately after performing the intermediate-language object 110 under management of the OPERETIGU system 203. An operating system 203 makes judgment that this is the file in which execution control is carried out by IPR111, from the file attribute of the intermediate-language object 110. An operating system 203 starts IPR111 as a result of this decision. (The system which has the interpreter system of the instruction for operating systems called a command shell apart from this approach can describe actuation similar as a text for command shells) .
[0095] On the other hand, IPR can acquire the argument passed from an operating system at the time of the starting. The file reference number (file management information) of the intermediate-language object file 110 is passed as this argument by the above-mentioned actuation. At this time, IPR loads the intermediate-language object 110 to that working area, and performs it. IPR111 started here serves as Master IPR below, and makes a part of delivery processing of a command share if needed to IPR112 which is SUREBU IPR.
[0096] The condition code of IPR in the condition of initial starting is a {busy}. Therefore, a demand is not received, even if other IPR(s) require a processing assignment of the parallel execution of processing and publish a {connection lock} message to this IPR. On the other hand, IPR which are operating state

other than initial starting can always operate as SUREBU IPR.
[0097] 3-3. execution-time processing of a juxtaposition description part -- next use drawing 18 and explain actuation of IPR at the time of parallel execution.
[0098] When the instruction word analysis processing 1801 of Master IPR detects instruction word {CALL addr (coexec)} during description of the intermediate-language object 110, the cobegin sentence processing 1802 is called. The cobegin sentence processing 1802 sends a {connection lock} command to all processors using a system bus 202. A response is a {busy}, or even if it passes through fixed time amount, when there is no response, it can judge that it is in the condition that these processors cannot share parallel execution. On the other hand, depending on whenever [allowances / of a processor resource], a processor while performing two or more IPR(s) exists, and two or more {acceptance} messages may be able to be received. Then, Master IPR creates the execution-time process control table 1701 in the working area. The contents of the execution-time process control table 1701 are the processor number and process number which were able to receive {acceptance} message, the block number of the processing (or it is due to carry out) block which considered that this process was SUREBU IPR and was distributed, and a pointer to the request queue 1807 to each process.
[0099] When the number of the blocks set as the object of parallel execution is smaller than the number of the processes of gained SUREBU IPR, Master IPR opens delivery connection for a {release} message to unnecessary SUREBU IPR. When there is more processing block count which should be carried out parallel execution contrary to this than gained SUREBU IPR, a queue 1807 is made to gained SUREBU IPR, and a processing block to carry out parallel execution to the element of this queue is added. This processing is processed by the scheduler 1803 which the cobegin sentence processing 1802 called. The new demand whose scheduler detected the schedule algorithm of this example is the simple algorithm of arranging to the queue 1807 of the shortest die length.
[0100] On the other hand, a part of processing block can be processed also in Master IPR. For this reason, a scheduler performs predicate expansion processing 1806. The predicate expansion processing 1806 takes out the list of a predicate called LODN fooCALL addr (coexec) of intermediate-language description, and transposes it to the list of a predicate called NOPCALL addr (foo). foo is the relative address of a certain procedure or a function here, and instruction word NOP expresses the instruction word {which does not act}. The instruction word description rewritten by the predicate expansion processing 1806 is again returned to the instruction word analysis processing 1801, and is evaluated there. For this reason, the instruction word analysis processing 1801 and the predicate expansion processing 1806 rewrite the instruction pointer 1708 of intermediate language according to the contents of processing.
[0101] In the above processing, it is determined by the scheduler 1803 in what kind of sequence two or more processing blocks set as the object of parallel execution are distributed to Master IPR and SUREBU IPR. Unless the scheduler 1803 of this example uses the special instruction mentioned later, at least one processing block performs assignment to Master IPR.
[0102] Master IPR carries out the ENQ of the {activation} message to the data stream according to the DS of a {load} message mentioned above in the queue 1807 over SUREBU IPR1808. SUREBU IPR which performs this processing demand returns the message which consists of DS 1510 after processing termination. {Dawn} will be returned if the condition code at this time is normal termination. On the other hand, when a certain run-time error is detected, a condition code {a fail} is returned. Although there is the need that the source code of an object program carries out a certain correspondence about error processing, this is the same as the conventional programming. The processing terminated normally by the condition code {Dawn} will return the value of a function with data 1512, if a processing block is a function.
[0103] The master IPR which started parallel execution waits for termination of parallel execution by the coend sentence. The already explained intermediate-language description which is generated to a coend sentence like is {CALL addr (sync)}. The instruction word analysis processing 1801 will call the sync sentence processing 1805, if this description is detected. The sync sentence processing 1805 checks coincidence with waiting, the processor number of the execution-time processor managed table 1701,

and a process number for the {Dawn} message from SUREBU IPR, and records processing termination. If processing termination of all processing blocks is checked, the sync sentence processing 1805 will be ended. Like, since [which was already described] Master IPR performs at least one processing block according to control of a scheduler 1803 and the IPR itself is a serial-processing system, it is that sync sentence processing 1805 is performed, after the activation of a processing block which Master IPR shared is completed.

[0104] Processing when access to a share variable occurs by monitor mold procedure call using execution-time processing of 3-4. shared resource access, next the flow chart of drawing 19 is explained. The instruction word analysis processing 1801 will take out the resource number of a monitor mold· from the maximum upper case of an IPR stack, if the call of procedure append or fetch is detected (S1901). The shared resource managed table 1001 is searched by using this resource number 1002 as a key (S1902). As a retrieval result, the block number 1004 with which the resource was declared is acquirable (S1903). As compared with the number of a self-block of this block number (S1904), when in agreement, the same processing as the usual procedure call is performed (S1907). If it is except it, IPR will access the element 502 of the execution-time block managed table 501 from the block number which a self-process is performing, and will follow the chain 508 from the chaining address 504 to a low order block (S1905). If the block which corresponds by this processing is found (S1906), procedure call processing will be performed (S1907). Even if it inspects till termination of a chain 508, when a block number in agreement cannot be detected, the processor number and process number which hold a monitor mold resource by the execution-time block management table are taken out, and a message {a fetch} or {appending} is stood by to it until it receives the data stream of a delivery (S1908) processing result to IPR specified by these two numbers (S1909).

[0105] On the other hand, in the IPR side which received this message, as shown in drawing 16 , the fetch processing S1613 or appending processing S1614 is performed. This processing takes out the resource number of a monitor mold shared resource from a part for the data division of a message, and inspects registration of the shared resource managed table 1001 according to this resource number. Next, the corresponding element is taken out, address calculation on the activation code of monitor mold procedure is performed from the DS 1005, actual processing is called, and a processing result is returned to a requestor side IPR with a condition code {Dawn}.

[0106] Here, since processing of IPR which receives a message and returns a response is serial processing when the monitor mold variable in other processors is accessed, it is completely controlled exclusively. Moreover, since IPR of a requestor side stands by, it can also prevent generating of the passing processing by the requestor side, until there is a response from IPR of other processors. Also when multiple processes are generated on the same processor and two or more IPR(s) are under activation, it limits to processing of dedication of access to a shared resource, and since activation of this processing is serially given to a target by IPR, it can control exclusively by the monitor mold.

[0107] Before an actual data stream is written in into a shared resource, even if it is going to read a value on the other hand, the contents of processing will become invalid. Moreover, the data to precede will be lost if it writes in the condition that all buffers were filled, further (multiplex writing). Although a synchronization mechanism is required to prevent these un-arranging, a synchronization is maintained by wait and signal processing to the queue described as a procedure inside a monitor mold.

[0108] By this example, activation of two or more IPR(s) is attained by the above processing, controlling exclusively.

[0109] 3-5. The processor explained beyond the procedure which specifies an activation processor was a processor which uses the same instruction word in after intermediate-language describing the processing block actually arranged at a remote processor, and the processing block which continues activation by the local processor. For this reason, the implementation approach of the processor which has the explicit **** need neither in the description which specifies a processor in intermediate-language description, nor the description for transmitting a processing block to the partial memory of a remote processor was explained. It is because this invention aimed at implementation of the processor which does not have the need for rewriting by the hardware change.

[0110] On the other hand, when a special purpose processor is added contrary to the above (it is especially important in specific fields, such as signal processing and an image processing), there is a case where he wants to specify clearly the processor which carries out processing distribution. For this reason, the processor of this example is function processor (slot:integer) as an intrinsic function. : boolean; was prepared. This is a function which returns logic [truth], when a processor substrate is mounted in the slot for the extended substrates and the intermediate-language interpreter is performed, if the number of the slot for the extended substrates of a system bus is specified by Argument slot (integer type). If this function is called, Master IPR will specify a processor number and will send a {connection lock} message. As for the return value of a function, {TRUE} will be substituted if there is a response of {acceptance} message as this response. Moreover, when there is no response beyond fixed time amount, or when a response is a {busy}, the return value of a function is {FALSE}.
[0111] Furthermore, the processor of this example prepared procedure distribute(slot:integer; procedure foo); as an inclusion procedure. This is a procedure which arranges and performs procedure which specifies the number of the slot for extended substrates by Argument slot (integer type), and is specified as the processor on the hardware of the slot by the procedure argument foo. The 2nd argument may be specified like function foo:tt;. foo is a function name and tt is a data type name here. For implementation of this procedure, if Master IPR specifies a processor number and has the response of {acceptance} message considering a {connection lock} message as delivery and this response, it will send {loading} and {activation} message continuously. By this processing, activation of the processing block specified by the procedure argument or functional parameter in the above-mentioned procedure is assigned to specific hardware, and is processed. The activation which specified the processor can be described using two above-mentioned inclusion processings. Next, it is an example of the description. It is the case where parallel execution of the procedure PROC1 with Arguments a and b and the procedure PROC2 with Argument a is carried out, and supposing he wants to arrange PROC1 to the processor of the slot 4 of a system bus, it can write as follows.
[0112] cobeginif processor(4) then distibute(4,PROC1(a,b))
else PROC1(a,b);
PROC2(a);
coend;
In this example, when there is no processor substrate in an expansion slot 4, parallel execution of the procedure PROC1 and PROC2 is carried out by the usual sequence.
[0113] 4. Summarize the role about the various table structures shown in explanation of this example using front drawing 20 which the supplementary 4-1. compiler of explanation and an interpreter use.
[0114] The block number managed table 801 is generated by the working area of a compiler 102, and when compile processing is completed, it is discarded. In the syntax-analysis means 104 of a compiler 102, the pointer of a block number and a block domestic disturbance table etc. is recorded on this table. Moreover, with reference to this front structure, chain generation during the block of the execution-time block managed table 501 is performed.
[0115] After the execution-time block managed table 501 is generated by the working area of a compiler 102, it is outputted to the file of the intermediate-language object 110 by the object file creation means 109. Furthermore, after the intermediate-language object 110 is loaded to an intermediate-language interpreter, it is referred to also within an intermediate-language interpreter. The analysis result of the argument list in the case of a block call and the dependency during a block are recorded on this table by the syntax-analysis means 104 of a compiler 102.
[0116] After the shared resource managed table 1001 is created by the working area of a compiler 102, it is outputted to the file of the intermediate-language object 110 by the object file creation means 109. This table is referred to also within an intermediate-language interpreter, after the intermediate-language object 110 is loaded to an intermediate-language interpreter. In an intermediate-language interpreter, it is referred to in monitor mold procedure processing.
[0117] The execution-time processor managed table 1701 is generated by the interpreter working area at the time of activation. This table is used for management of an usable (the {connection lock} was able to

be carried out) processor, and schedule pipe ** of the result which carried out parallel execution in parallel execution.
[0118] In addition to the above-mentioned table structure, by this example, the identifier table and the block domestic disturbance table were used for management of identifiers, such as a variable name and a block name, in the compiler 102.
[0119] 4-2. In expansion 4-2-1. exclusive control of this example, and explanation of the shared resource management above, the monitor mold was mentioned and explained as the exclusive control to a shared resource, and a synchronization mechanism. this -- this example -- setting -- implementation -- it was used as an easy thing. However, in the same configuration as this example, it is applicable also to a well-known semaphore immediately. What is necessary is just for that, to record under management of which block a semaphore is accessed by the same DS as the execution-time block managed table 501 of this example, and to prepare the inclusion procedure which accesses RIMOTO with a communication link about the semaphore besides intraprocessor.
[0120] Well-known exclusive control and a synchronization mechanism can be replaced using a semaphore in many cases. This example can be said [that it is applicable also to the share variable implementation approaches other than a monitor mold, and] from this.
[0121] Moreover, in the monitor mold shown by this example, when the structure which removed buffer memory assignment is used, it can be considered that the synchronous communication path in a specific procedure connected to a certain data type was realized by the intermediate-language interpreter. Therefore, the intermediate-language interpreter of this invention can be mounted also to the case where a communication path is realized using memory structures, such as FIFO, between interpreters. In this case, it does not have a share variable, but even if it is the compiler of the programming language of the specification which performs data transfer during parallel execution procedure by communication link, it can say [that it can perform and] in the intermediate-language interpreter mounted in two or more processors by performing the same intermediate-language output as this example.
[0122] 4-2-2. An intermediate-language interpreter equivalent to what the mounting above-mentioned example of an intermediate-language interpreter showed may be realized by the processor which consists of two or more processors. For example, the kernel program which realizes parallel processing is mounted on the computer which consists of four processors, and it is possible to run an intermediate-language interpreter program on this kernel program. If this is developed further, the intermediate-language interpreter of this invention can be mounted on the separate multiprocessor system with which topology differs, respectively, and the juxtaposition program which can be performed in a topology mixture environment can be described.
[0123] If this approach is used, the high level language which can be described on the system which combined the systolic array processor system advantageous to high-speed processing of an image processing and the vector processor system advantageous to processing of signal processing, high-speed fourier transform processing, etc. is realizable.
[0124] The output code of the compiler developed considering use of a vector processor as a premise in the conventional compiler generates the machine language code performed on a vector processor. This object code is a code depending on the target system. Unless it recompiles the output of this kind of compiler to modification of the specification of hardware, it does not serve as the optimum code. For example, even if it newly adds a vector processor to the computer which was employing the object code compiled considering the hardware of only a scalar processor as an execution environment, unless it compiles again and regenerates object code with the compiler which supports a vector processor, the improvement effectiveness of execution speed is not acquired.
[0125] On the other hand, the compiler shown by this example generates the code performed by the pseudo code interpreter. Moreover, the schedule of the pseudo code is dynamically carried out to a processor at the time of activation. When the hardware (circuit board etc.) which mounted the pseudo code interpreter from these two descriptions to the system which is employing object code [finishing / compile] is added, the procedure which compiles again and regenerates the object code of an object program is unnecessary. Object code of an object program is performed from the time of the hardware

which mounted the pseudo code interpreter being added by the multiprocessor system by the processor of addition hardware, and the processor which was being employed from the former. This is very advantageous in the phase of selling and employing commercial application. Because, in order for most commercial applications to protect copyright to program, it is because it is sold to a user only in the state of the object code which does not include a source code. If it is the object code developed by the compiler of this example, even if a user performs hardware addition, there will be no need of changing the object code of application software. Thereby, the effectiveness of excelling in both sides of the ease of maintenance and the simplicity of employment arises.

[0126]

[Effect of the Invention] The device which controls exclusively access to the variable with which this invention is shared at the time of parallel execution, The compiler which generates the object code described by the instruction word of intermediate language, Since it is constituted by the interpreter which performs intermediate language which said compiler outputted, and the device in which processing block assignment of the parallel execution realized by said interpreter is performed dynamically, It is possible to perform parallel processing, without recompiling the object code developed using this compiler, even when the configuration of the hardware of a parallel processing system changes.

[0127] Even when addition hardware, such as an accelerator, is added since a user is a processing speed improvement if it puts in another way, about the application program for the personal computers which mounted the processor by this invention, an improvement of the processing speed by the hardware added without needing modification of the program by hardware addition can be enjoyed.

[Translation done.]

* NOTICES *

1.This document has been translated by computer. So the translation may not reflect the original precisely.
2.**** shows the word which can not be translated.
3.In the drawings, any words are not translated.

CLAIMS

[Claim(s)]
[Claim 1] The multiprocessor processor carry out being constituted by the compile device have a means analyze the programme description which generates access to shared memory resource assignment at least, and direct exclusive control, and a means analyze the description to the program-manipulation unit in which parallel processing is possible, and perform processor assignment, the interpreting device perform the intermediate language which said compile device outputted, and the device carry out dynamically the processing block assignment of the parallel execution realized according to said interpreting device as the description.

[Translation done.]

* NOTICES *

1.This document has been translated by computer. So the translation may not reflect the original precisely.
2.**** shows the word which can not be translated.
3.In the drawings, any words are not translated.

DESCRIPTION OF DRAWINGS

[Brief Description of the Drawings]
[Drawing 1] A multiprocessor processor suitable as an example of this invention, and the block diagram of the compiler processor.
[Drawing 2] The block diagram of the personal computer which is the multiprocessor processor which performs object code developed by the compiler processor of drawing 1 .
[Drawing 3] The explanatory view of a monitor mold.
[Drawing 4] The flow chart of processing of the syntax-analysis means 104.
[Drawing 5] The explanatory view of an execution-time block managed table.
[Drawing 6] The explanatory view of the program mentioned as an example.
[Drawing 7] The explanatory view of the dependency of a block of the program mentioned as an example.
[Drawing 8] The explanatory view of a block managed table.
[Drawing 9] The flow chart of the processing which takes out the dependency during a block.
[Drawing 10] The explanatory view of a shared resource managed table.
[Drawing 11] The symbol description Fig. of intermediate language.
[Drawing 12] The explanatory view of processing of intermediate-language generating.
[Drawing 13] The explanatory view of an intermediate-language generating procedure.
[Drawing 14] The explanatory view of the stack condition at the time of intermediate-language processing.
[Drawing 15] The explanatory view of the DS used for the communication link between the processes which form an intermediate-language interpreter.
[Drawing 16] The flow chart of processing of an intermediate-language interpreter.
[Drawing 17] The block diagram of the process which realizes an intermediate-language interpreter.
[Drawing 18] A cobegin sentence, the explanatory view of sync sentence processing.
[Drawing 19] The flow chart at the time of activation of shared resource access processing.
[Drawing 20] The explanatory view of front structure.
[Description of Notations]
101 -- Source code
102 -- Compiler
103 -- Lexical-analysis means
104 -- Syntax-analysis means
105 -- Shared memory resource detection means
106 -- Parallel execution detection means
107 -- Code generation means
108 -- Code generation means
109 -- Object file creation means
110 -- Intermediate-language object
111 -- Intermediate-language interpreter

112 -- Intermediate-language interpreter
113 -- Processor
115 -- Communication path
200 -- Personal computer
201 -- Addition hardware
202 -- System bus
203 -- Operating system
300 -- Source code for explanation
501 -- Execution-time block managed table
503 -- Block number
504 -- The chaining address to a low order block
507 -- The chaining address to an argument list
508 -- Chain
509 -- Chain
801 -- Block number managed table
802 -- Block number
803 -- Block level
804 -- Table entry
1001 -- Shared resource managed table
1002 -- Resource number
1003 -- Share variable name
1004 -- Block number with which the shared resource was declared
1006 -- Activation code of monitor mold procedure
1203 -- Argument list
1401 -- Hysteresis of stack use
1501 -- DS
1502 -- Command
1503 -- Processor number
1504 -- Process number
1506 -- Data length
1507 -- Object code
1508 -- Stack initialization data
1509 -- Table initialization data
1511 -- Condition code
1701 -- Execution-time processor managed table
1702 -- Process
1706 -- Coding region
1707 -- Stack area
1708 -- Instruction pointer
1709 -- Stack pointer
1801 -- Instruction word analysis processing
1803 -- Scheduler
1804 -- Predicate expansion processing
An intermediate-language interpreter besides 1808 --

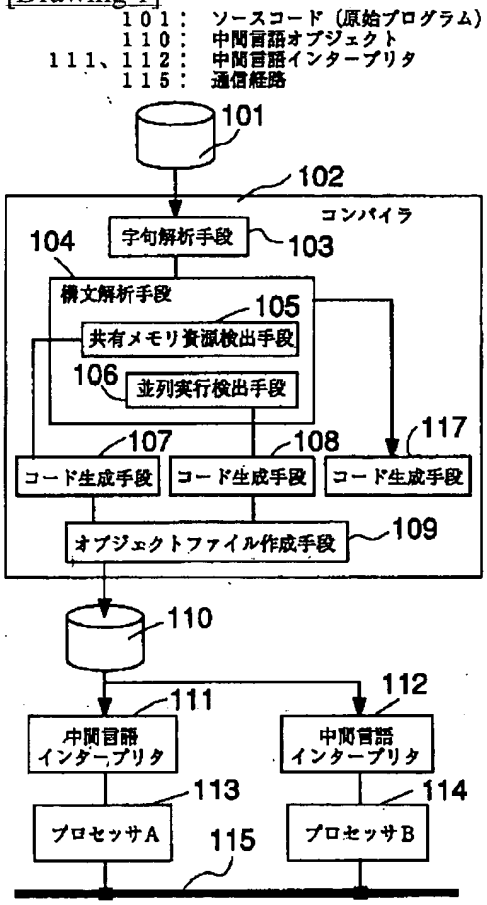
[Translation done.]

* NOTICES *

1.This document has been translated by computer. So the translation may not reflect the original precisely.
2.**** shows the word which can not be translated.
3.In the drawings, any words are not translated.

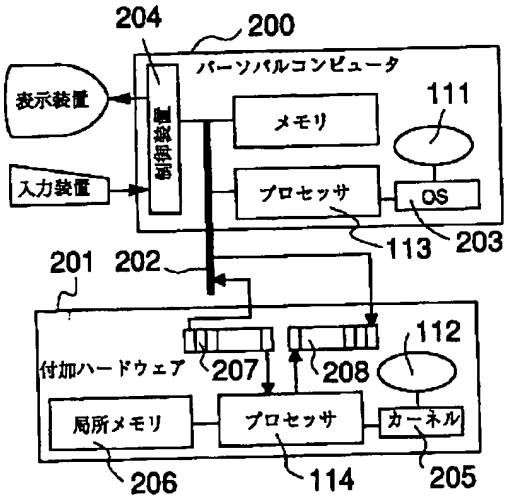
DRAWINGS

[Drawing 1]

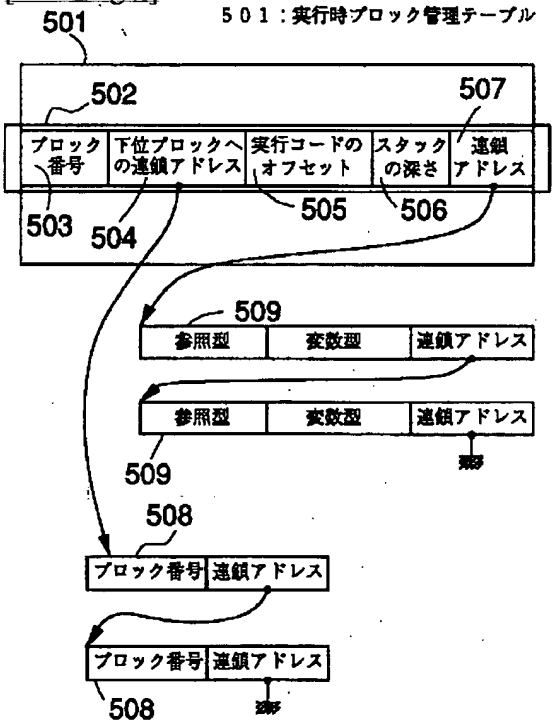


[Drawing 2]

200：パーソナルコンピュータ
111、112：中間言語インタープリタ
201：付加ハードウェア
202：システムバス

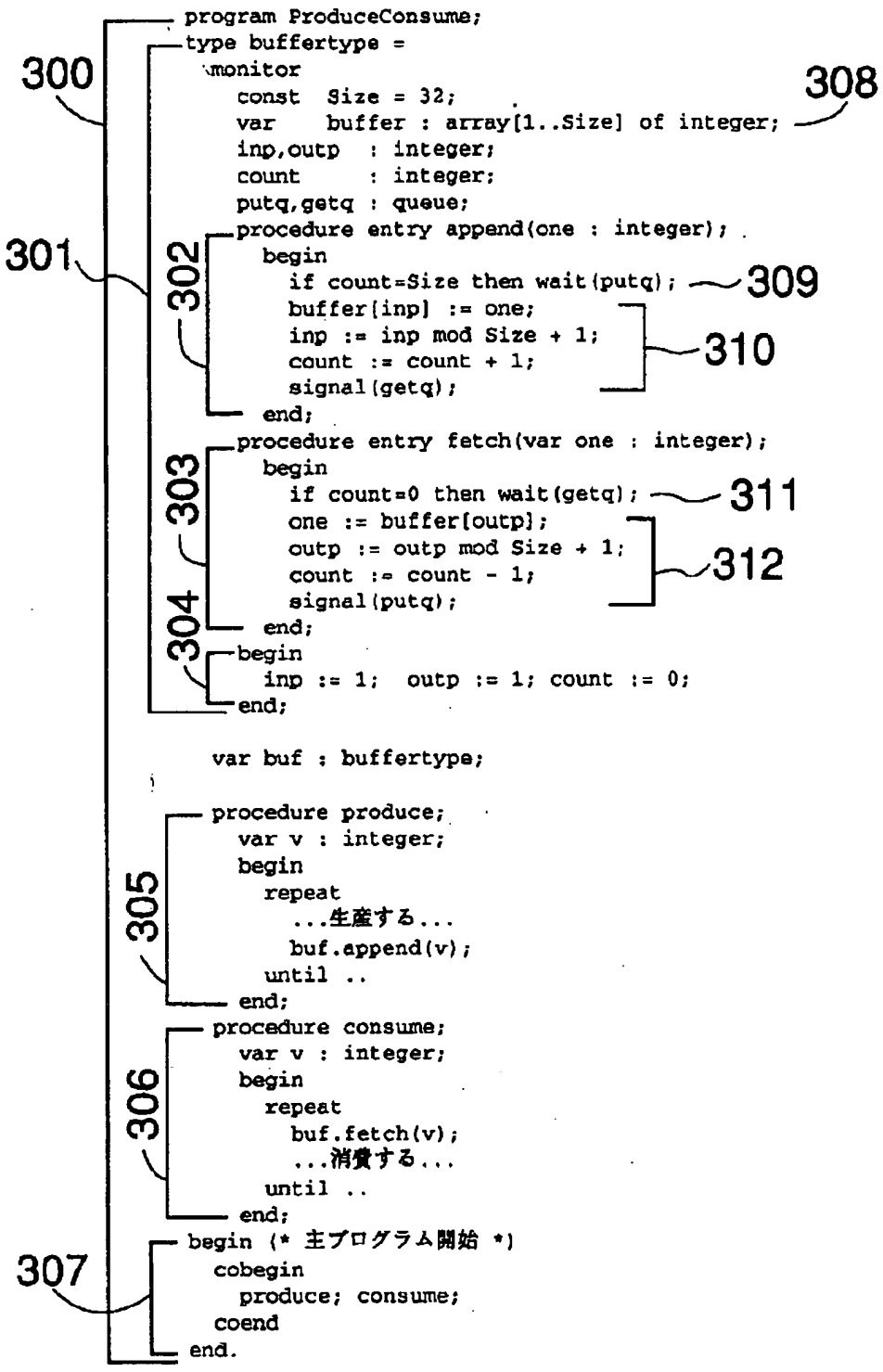


[Drawing 5]

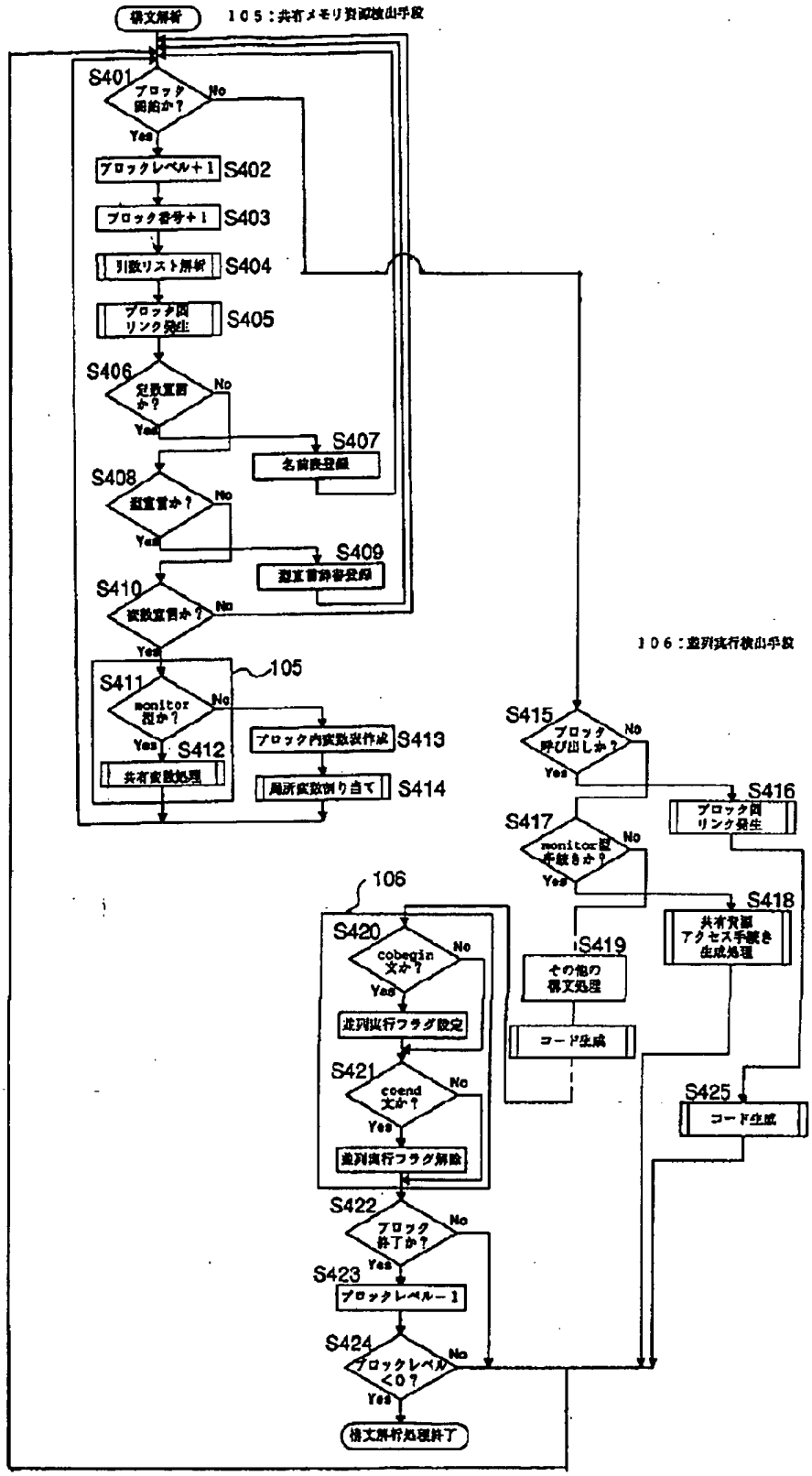


[Drawing 3]

３０１：モニタ型の型宣言



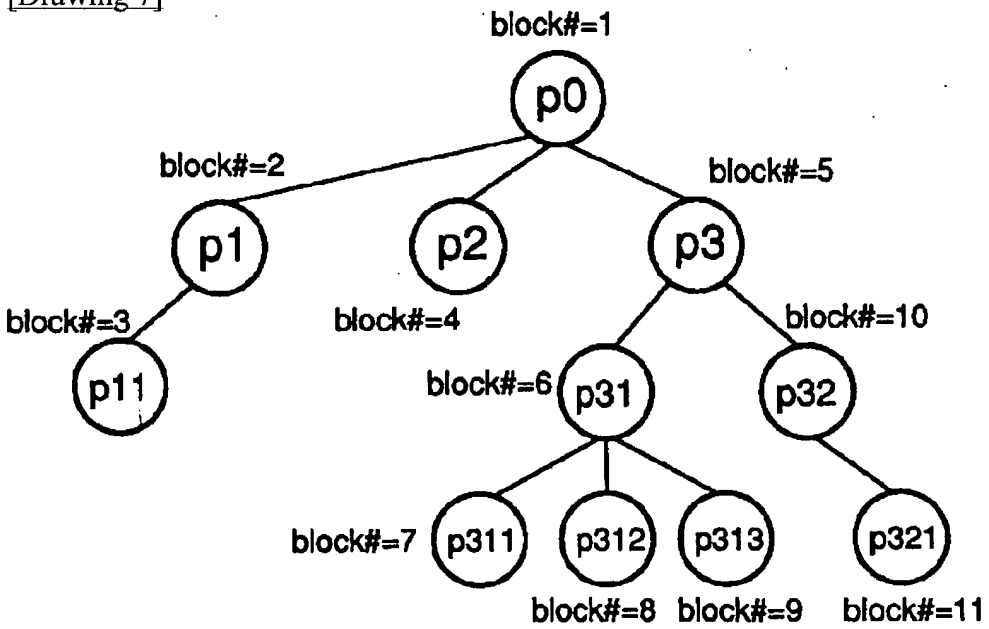
[Drawing 4]


構文解析
105：共有メモリ変数検出手段
S401
ブロック開始か？
No
Yes
ブロックレベル＋1
S402
ブロック番号＋1
S403
引数リスト解析
S404
ブロック図リンク発生
S405
S406
定数宣言か？
No
Yes
S407
名前表登録
S408
変数宣言か？
No
Yes
S409
変数情報登録
S410
配列宣言か？
No
Yes
105
S411
monitor宣か？
No
Yes
S412
共有変数処理
ブロック内変数表作成
S413
局所変数割り当て
S414
106：並列実行検出手段
S415
ブロック呼び出しか？
No
Yes
S416
ブロック図リンク発生
S417
monitor宣言終了か？
No
Yes
S418
共有資源アクセス手続き生成処理
S419
その他の構文処理
コード生成
106
S420
cobegin文か？
No
Yes
並列実行フラグ設定
S421
coend文か？
No
Yes
並列実行フラグ解除
S422
ブロック終了か？
No
Yes
S423
ブロックレベル－1
S424
ブロックレベル＜0？
No
Yes
構文解析処理終了
S425
コード生成

[Drawing 6]

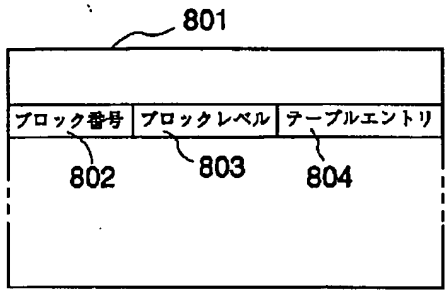
```
program p0;

  procedure p1;
    function p11: integer;
      begin ..... end;
   begin
     ....;
   end;

  procedure p2;
   begin ..p1;... end;

  procedure p3;
    procedure p31;
        procedure p311;
             begin .... end;
        procedure p312;
             begin .... end;
        function p313: integer;
             begin .... end;
     begin ... end;
    procedure p32;
        procedure p321;
             begin .... end;
     begin ... end;
   begin
     ....
   end;

begin
  .... p3; .....;
end.
```

[Drawing 7]

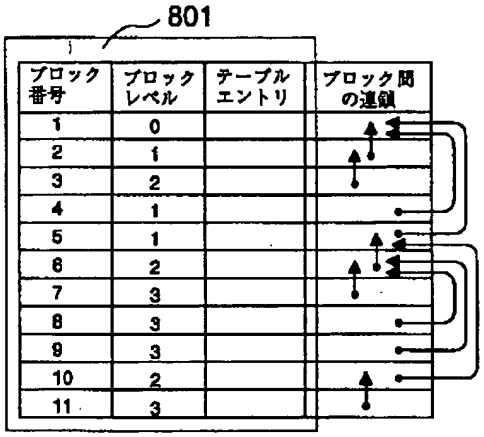


[Drawing 8]

801：ブロック番号管理テーブル

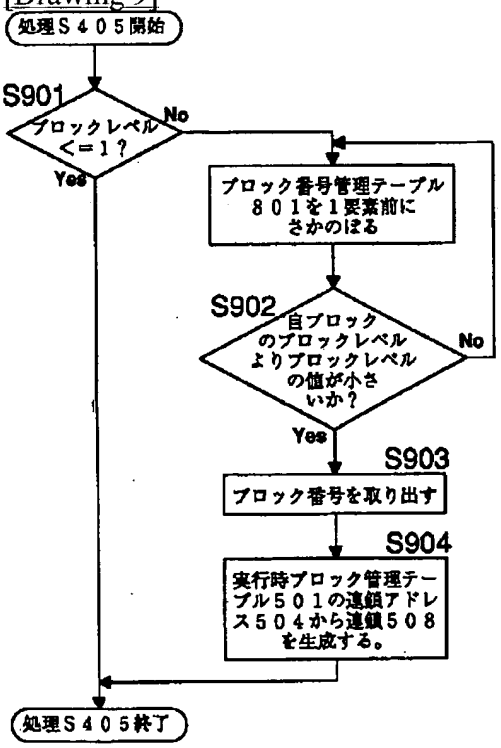


(a)

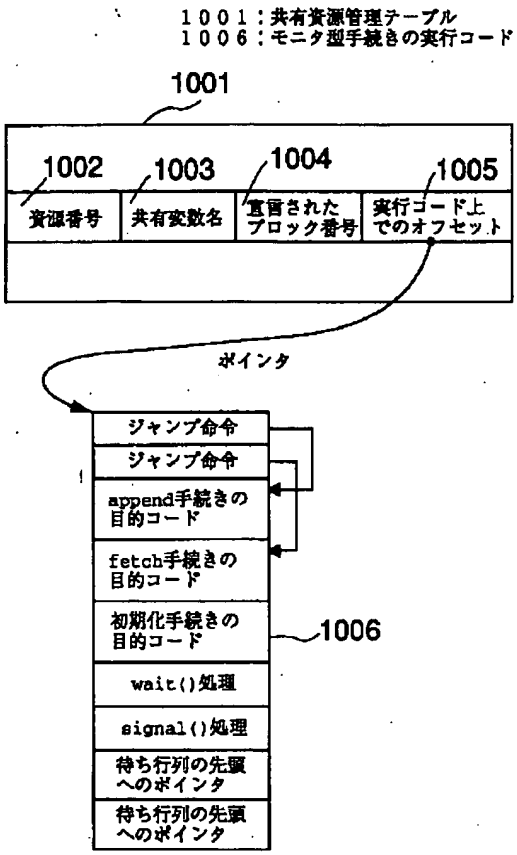


(b)

[Drawing 9]



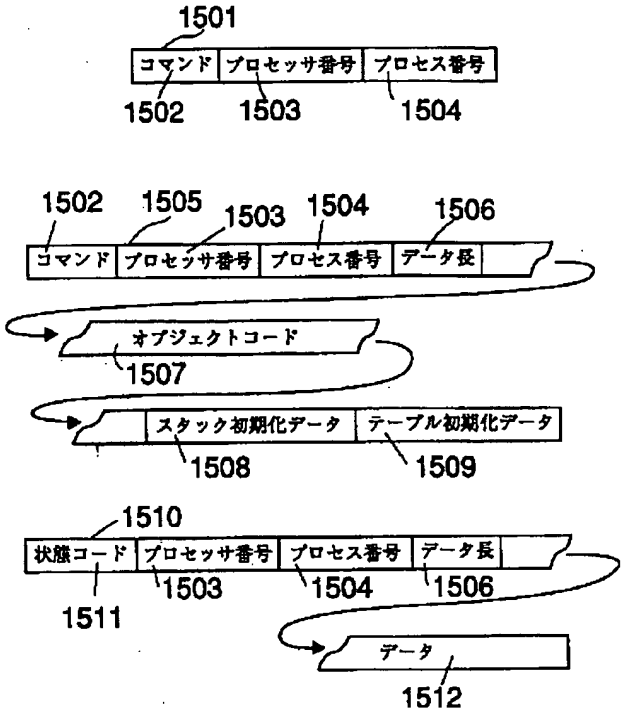
[Drawing 10]



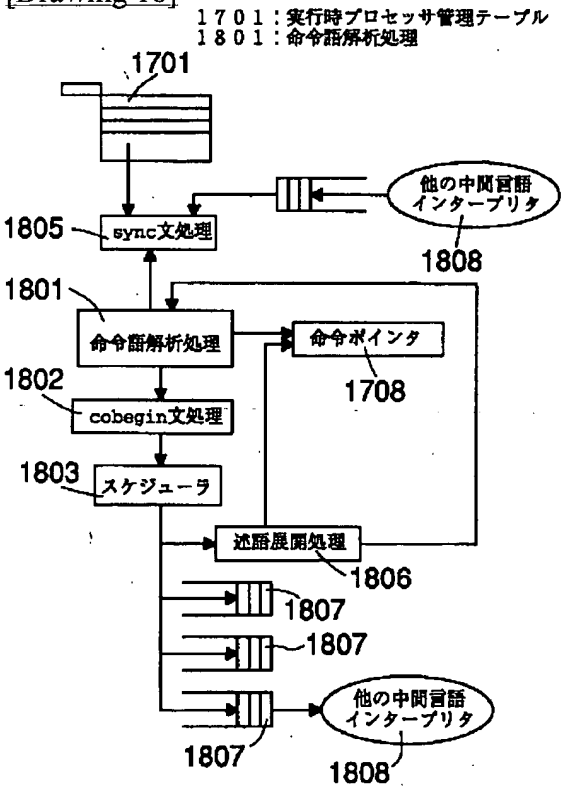
[Drawing 11]

命令語	名前の意味	動作の概要
LODV	Load Value	(TOP-1) ← value SP ← SP-1
LODA	Load Address	(TOP-1) ← addr(x) SP ← SP-1
LODN	Load Number of an object	find number of an obj. (TOP-1) ← number of an obj
LOD n	Load object	(TOP-1) ← (TOP+n) SP ← SP-1
ALOC n	Alocate	SP ← SP-n
UALC n	UnAlocate	SP ← SP+n
CALL	Call subroutine	ar[1] ← PC+1 PC ← addr(x)
RET	return from ..	PC ← ar[1]
ADD	Addition	(TOP) ← (TOP+1)+(TOP)
RELS	Release	release connection

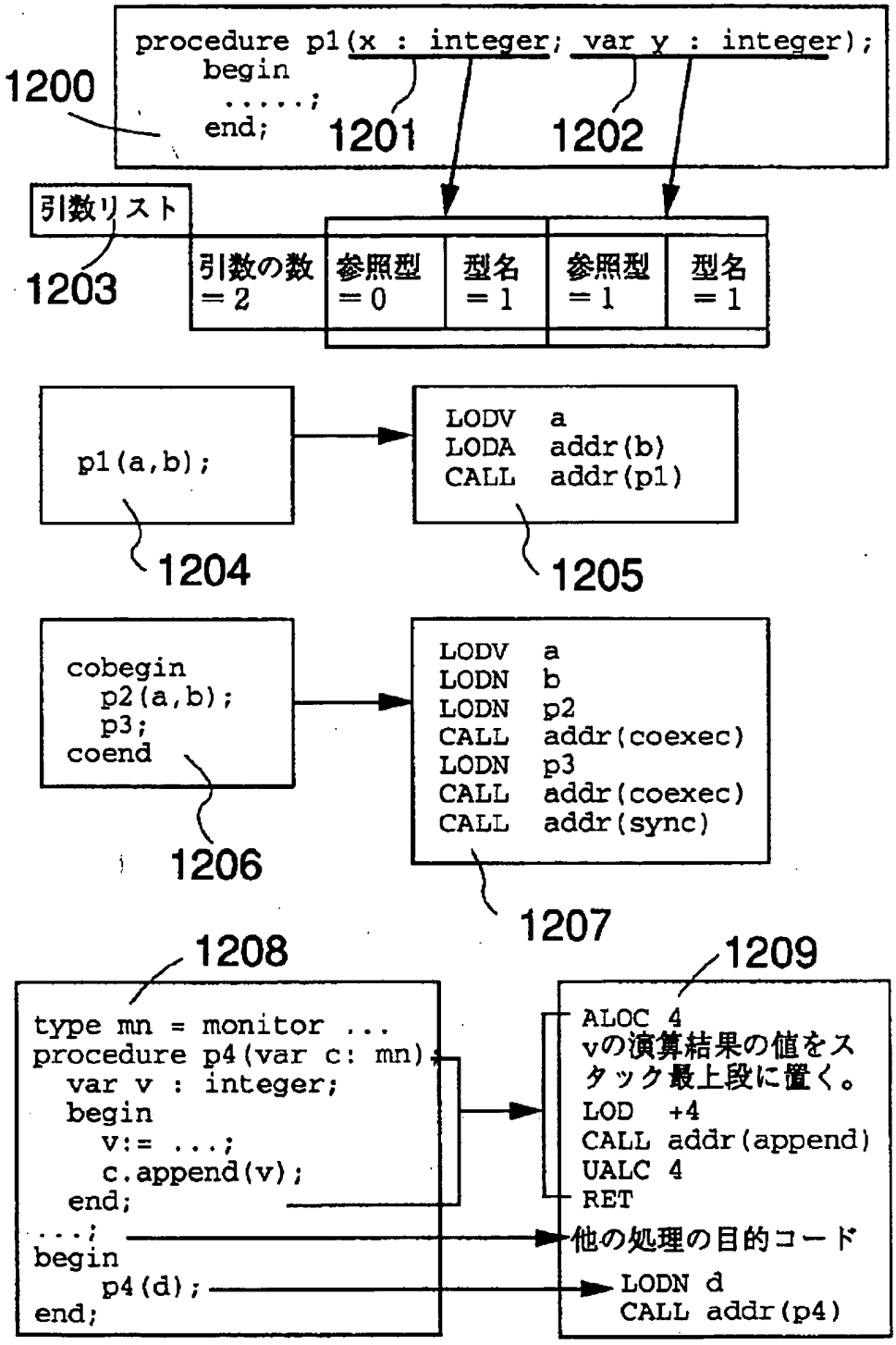
[Drawing 15]



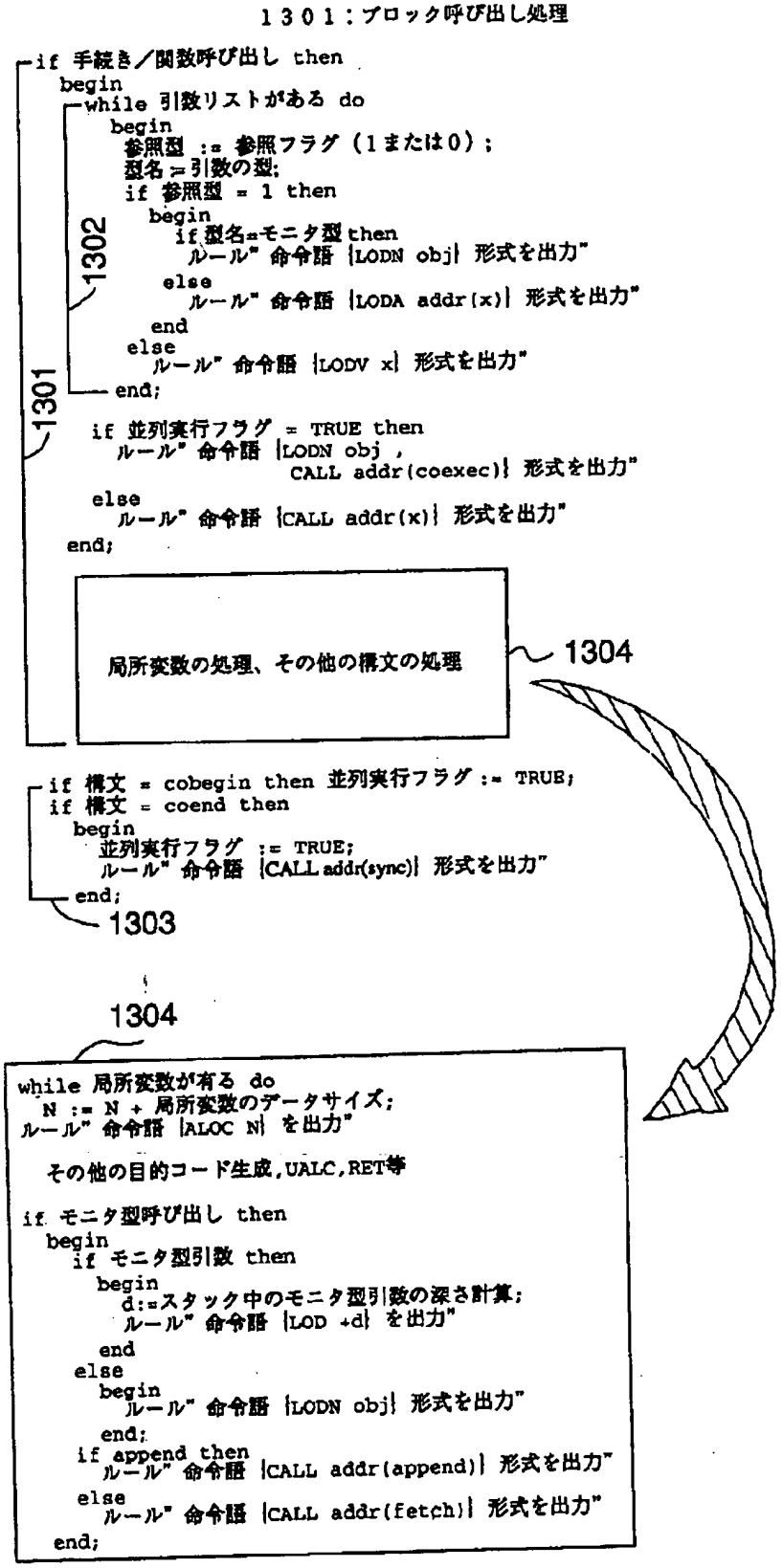
[Drawing 18]



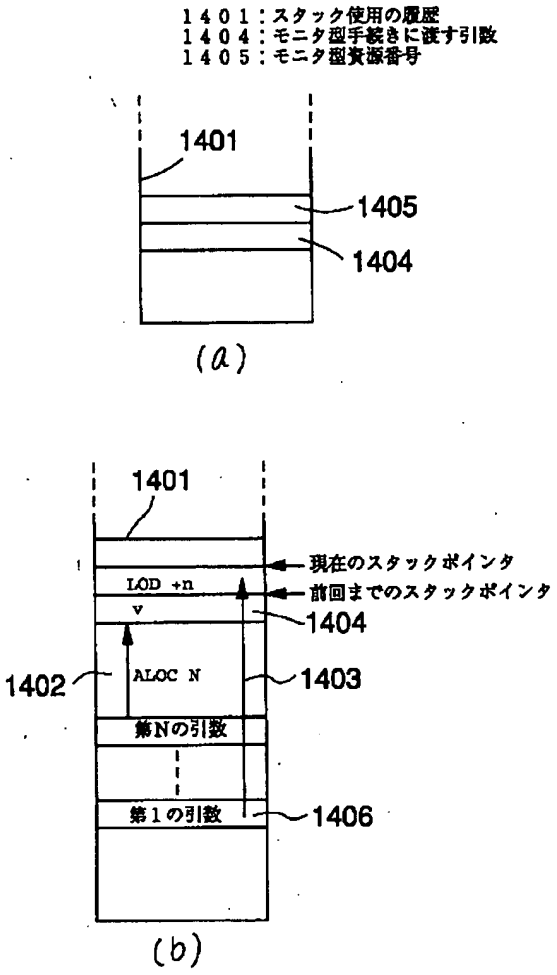
[Drawing 12]


procedure p1(x : integer; var y : integer);
begin
.....;
end;
1200
1201
1202
引数リスト
1203
引数の数 ＝2
参照型 ＝0
型名 ＝1
参照型 ＝1
型名 ＝1
p1(a,b);
1204
LODV a
LODA addr(b)
CALL addr(p1)
1205
cobegin
p2(a,b);
p3;
coend
1206
LODV a
LODN b
LODN p2
CALL addr(coexec)
LODN p3
CALL addr(coexec)
CALL addr(sync)
1207
1208
type mn = monitor ...
procedure p4(var c: mn);
var v : integer;
begin
v:= ...;
c.append(v);
end;
...;
begin
p4(d);
end;
1209
ALOC 4
vの演算結果の値をスタック最上段に置く。
LOD +4
CALL addr(append)
UALC 4
RET
他の処理の目的コード
LODN d
CALL addr(p4)

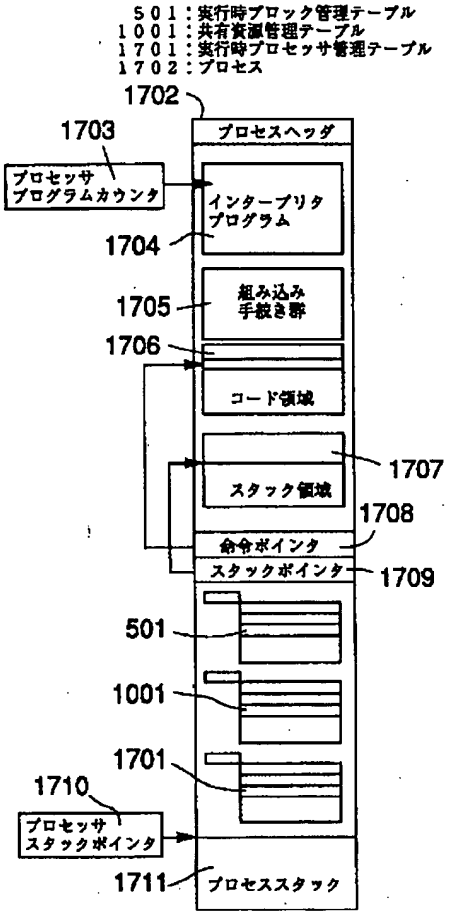
[Drawing 13]


1301：ブロック呼び出し処理
if 手続き／関数呼び出し then
begin
while 引数リストがある do
begin
参照型 := 参照フラグ（1または0）;
型名:=引数の型;
if 参照型 = 1 then
begin
if 型名=モニタ型 then
ルール" 命令語 |LODN obj| 形式を出力"
else
ルール" 命令語 |LODA addr(x)| 形式を出力"
end
else
ルール" 命令語 |LODV x| 形式を出力"
end;
1302
1301
if 並列実行フラグ = TRUE then
ルール" 命令語 |LODN obj ,
CALL addr(coexec)| 形式を出力"
else
ルール" 命令語 |CALL addr(x)| 形式を出力"
end;
局所変数の処理、その他の構文の処理
1304
if 構文 = cobegin then 並列実行フラグ := TRUE;
if 構文 = coend then
begin
並列実行フラグ := TRUE;
ルール" 命令語 |CALL addr(sync)| 形式を出力"
end;
1303
1304
while 局所変数が有る do
N := N + 局所変数のデータサイズ;
ルール" 命令語 |ALOC N| を出力"
その他の目的コード生成,UALC,RET等
if モニタ型呼び出し then
begin
if モニタ型引数 then
begin
d:=スタック中のモニタ型引数の深さ計算;
ルール" 命令語 |LOD +d| を出力"
end
else
begin
ルール" 命令語 |LODN obj| 形式を出力"
end;
if append then
ルール" 命令語 |CALL addr(append)| 形式を出力"
else
ルール" 命令語 |CALL addr(fetch)| 形式を出力"
end;

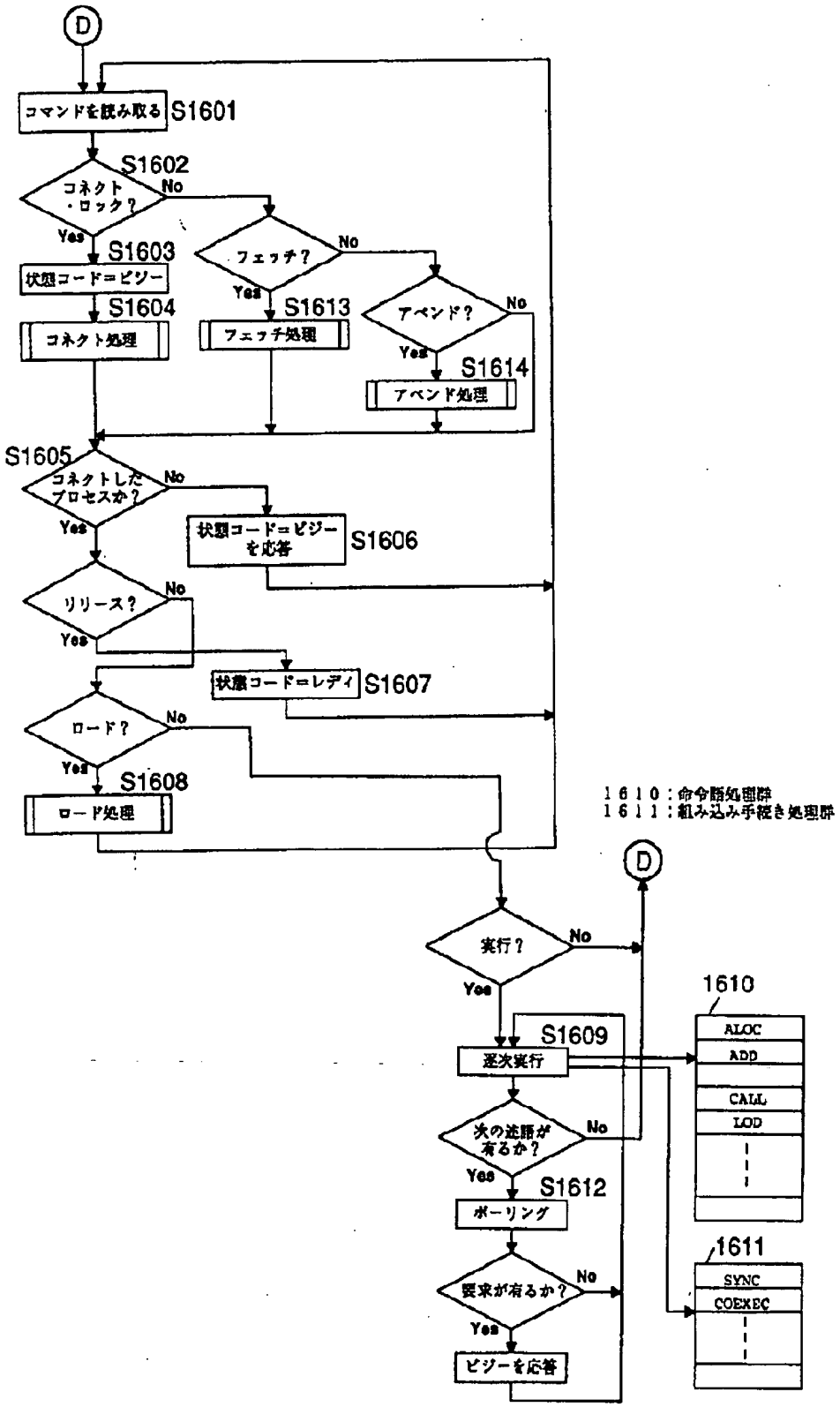
[Drawing 14]


1401：スタック使用の履歴
1404：モニタ型手続きに渡す引数
1405：モニタ型資源番号
1401
1405
1404
(a)
1401
現在のスタックポインタ
LOD +n
前回までのスタックポインタ
v
1404
1402
ALOC N
1403
第Nの引数
第1の引数
1406
(b)

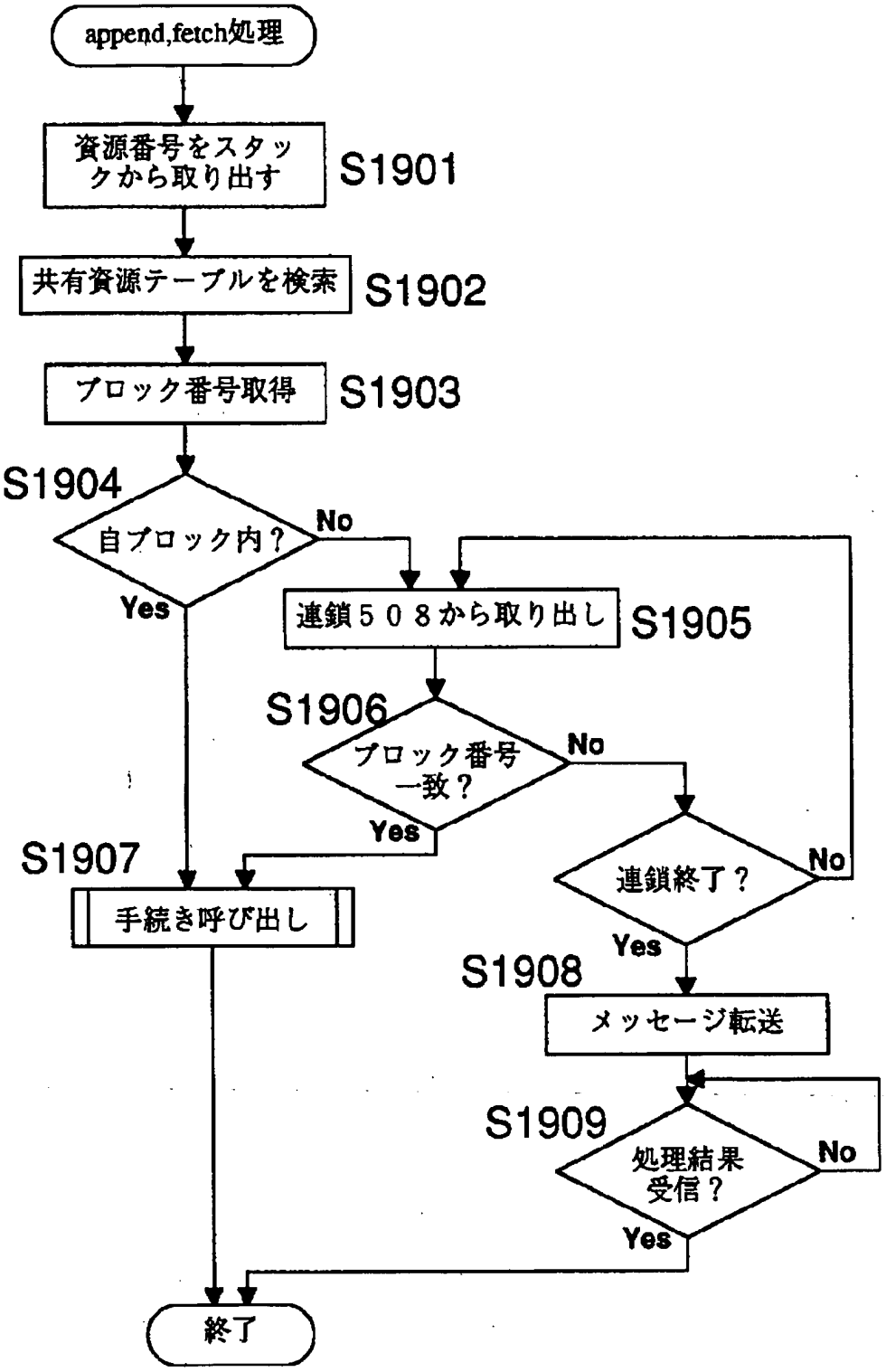
[Drawing 17]


501：実行時ブロック管理テーブル
1001：共有資源管理テーブル
1701：実行時プロセッサ管理テーブル
1702：プロセス
1702
1703
プロセスヘッダ
プロセッサ
プログラムカウンタ
インタープリタ
プログラム
1704
組み込み
手続き群
1705
1706
コード領域
1707
スタック領域
1708
命令ポインタ
スタックポインタ
1709
501
1001
1701
1710
プロセッサ
スタックポインタ
1711
プロセススタック

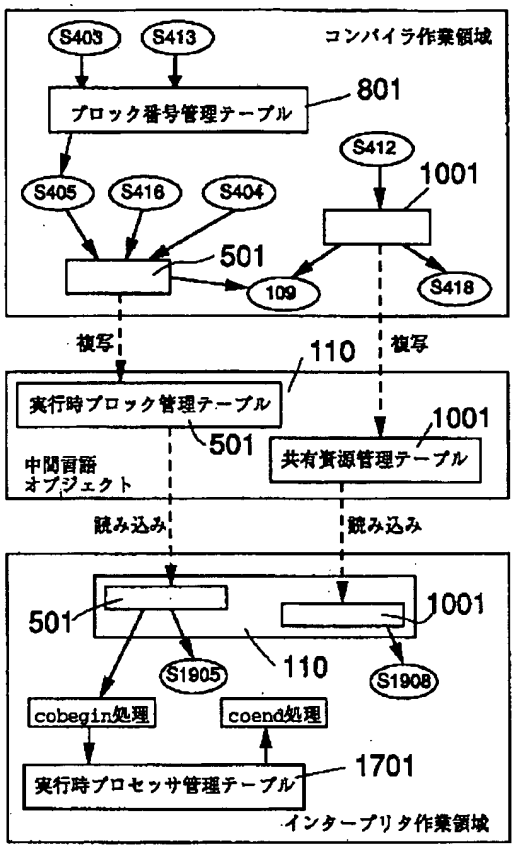
[Drawing 16]


D
コマンドを読み取る
S1601
S1602
コネクト・ロック?
No
Yes
S1603
状態コード=ビジー
S1604
コネクト処理
フェッチ?
No
Yes
S1613
フェッチ処理
アペンド?
No
Yes
S1614
アペンド処理
S1605
コネクトしたプロセスか?
No
Yes
状態コード=ビジーを応答
S1606
リリース?
No
Yes
状態コード=レディ
S1607
ロード?
No
Yes
S1608
ロード処理
1610:命令語処理群
1611:組み込み手続き処理群
D
実行?
No
Yes
1610
S1609
逐次実行
ALOC
ADD
CALL
LOD
次の述語が有るか?
No
Yes
S1612
ポーリング
1611
要求が有るか?
No
Yes
SYNC
COEXEC
ビジーを応答

[Drawing 19]


append,fetch処理
資源番号をスタックから取り出す
S1901
共有資源テーブルを検索
S1902
ブロック番号取得
S1903
S1904
自ブロック内？
No
Yes
連鎖５０８から取り出し
S1905
S1906
ブロック番号一致？
No
Yes
S1907
手続き呼び出し
連鎖終了？
No
Yes
S1908
メッセージ転送
S1909
処理結果受信？
No
Yes
終了

[Drawing 20]



[Translation done.]

(19)日本国特許庁（ＪＰ）　(12) **公開特許公報**（Ａ）　(11)特許出願公開番号

特開平6－4498

(43)公開日　平成6年(1994)1月14日

(51)Int.Cl.[5]		識別記号	庁内整理番号	ＦＩ		技術表示箇所
Ｇ０６Ｆ	15/16	４３０	9190－5L			
	9/45					
			9292－5B	Ｇ０６Ｆ　9/44	３２２　Ｆ	

審査請求　未請求　請求項の数1(全 26 頁)

(21)出願番号　特願平4－162892

(22)出願日　平成4年(1992)6月22日

(71)出願人　000002369
セイコーエプソン株式会社
東京都新宿区西新宿2丁目4番1号

(72)発明者　長坂　文夫
長野県諏訪市大和3丁目3番5号　セイコーエプソン株式会社内

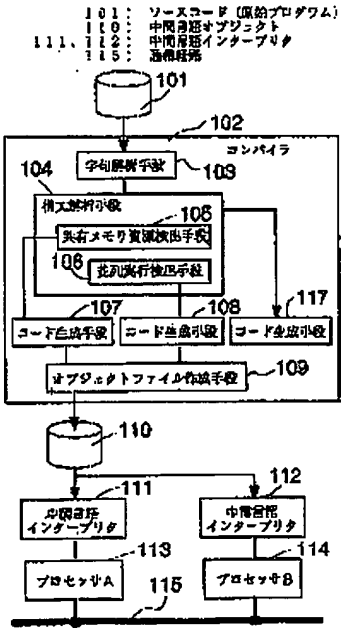
(74)代理人　弁理士　鈴木　喜三郎　(外1名)

(54)【発明の名称】　マルチプロセッサ処理装置

(57)【要約】

【目的】　並列処理系のハードウェアの構成が変わった場合でも、オブジェクトコードを再コンパイルする事無くマルチプロセッサによる並列処理を可能にすること。

【構成】　少なくとも共有メモリ資源割り当てに対してアクセスを発生するプログラム記述の解析を行ない排他制御を指示する手段と、並列処理可能なプログラム処理単位に対する記述の解析を行ないプロセッサ割り当てを行なう手段とを有するコンパイル機構と、前記コンパイル機構の出力した中間言語を実行するインタープリット機構と、前記インタープリット機構により実現される並列実行の処理ブロック割り当てを動的に行なう機構とによって構成されている。



1

【特許請求の範囲】

【請求項１】　少なくとも共有メモリ資源割り当てに対してアクセスを発生するプログラム記述の解析を行ない排他制御を指示する手段と、並列処理可能なプログラム処理単位に対する記述の解析を行ないプロセッサ割り当てを行なう手段とを有するコンパイル機構と、前記コンパイル機構の出力した中間言語を実行するインタープリット機構と、前記インタープリット機構により実現される並列実行の処理ブロック割り当てを動的に行なう機構とによって構成されることを特徴とするマルチプロセッサ処理装置。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】本発明はアーキテクチャの異なるマルチプロセッサによる並列実行処理に関する。

【０００２】

【従来の技術】同一のプロセッサを複数用いることで均質な並列処理を行なうシステムは既に知られている。

【０００３】一方、異なるアーキテクチャを持つプロセッサを複数用いることで並列処理を行なうシステムについては、コンパイルにより直接機械語に落とす方法は知られている。例えば、特開昭６２－３４２７５号で述べられている大型コンピュータのベクトルプロセッサ装置などはこれに該当する。しかし、この方法では個々のアーキテクチャの差を吸収し、均質なマルチプロセッサ処理系と等価な実行環境を実現することはできなかった。

【０００４】そこで、その問題を解決する方法としてコンパイルにより直接機械語に落とさずに、その間に中間コードを介在させる方法が特開昭６３－４１９３４号で提案されている。この方法では中間言語オブジェクトコードを発生するコンパイラと中間言語インタープリタにより処理が行なわれ、それによりＣＰＵと外部メモリ間のアクセス頻度を減少させている。

【０００５】

【発明が解決しようとする課題】しかし、上記従来発明は並列処理に必要な共有変数の排他制御、実行単位の分散を実現するものではなかったため、並列記述言語によって書かれた目的プログラムが実際には並列実行されないという問題点があった。

【０００６】本発明はこの様な問題を解決するために鑑みられたもので、その目的とするところは、異なるアーキテクチャを持つプロセッサを複数用いることで並列処理を行なうシステムにおいて、並列記述言語仕様に基づいて書かれた目的プログラムにより並列実行を実現することと、プロセッサユニットの変更があった場合でも目的プログラムの変更を行ない、再コンパイルするという一連の作業なく並列実行を可能にすることにある。

【０００７】

【課題を解決するための手段】この様な課題を解決するために本発明のマルチプロセッサ処理装置は、少なくとも共有メモリ資源割り当てに対してアクセスを発生するプログラム記述の解析を行ない排他制御を指示する手段と、並列処理可能なプログラム処理単位に対する記述の解析を行ないプロセッサ割り当てを行なう手段とを有するコンパイル機構と、前記コンパイル機構の出力した中間言語を実行するインタープリット機構と、前記インタープリット機構により実現される並列実行の処理ブロック割り当てを動的に行なう機構とによって構成されている。

2

【０００８】

【実施例】実施例の説明は次の各項目に従って行なう。

【０００９】０．はじめに

１．ソースコードのコンパイル

１－１．ターゲットシステムの説明

１－２．並列実行及び同期、排他制御の記述方法

１－３．字句解析、構文解析の処理

１－３－１．引数リスト解析処理（Ｓ４０４）

１－３－２．ブロック間の従属関係の抽出（Ｓ４０５、Ｓ４１６）

１－３－３．共有変数処理（Ｓ４１２）

２．中間言語によるコード生成

２－１．並列実行時のコード生成手段１０８

２－２．共有資源アクセス手続きのコード生成手段（Ｓ４１８）

２－３．オブジェクトファイル作成手段１０９の動作

３．中間言語インタープリタ

３－１．中間言語インタープリタ間の通信

３－２．中間言語インタープリタの動作状態

３－３．並列記述部分の実行時処理

３－４．共有資源アクセスの実行時処理

３－５．実行プロセッサを明示する手続き

４．説明の補足

４－１．コンパイラ、インタープリタの使用する表

４－２．本実施例の展開

４－２－１．排他制御と共有資源管理

４－２－２．中間言語インタープリタの実装

０．はじめに

図１は本発明の一実施例として好適なマルチプロセッサシステムとそのコンパイラの構成図である。ソースコード１０１のコンパイルは、ターゲットシステムと異なるコンピュータ上で行なわれても良い。この時、ソースコード１０１を入力として、中間言語オブジェクト１１０が出力される。この中間言語オブジェクト１１０が実際のターゲットシステムにおいて実行される。

【００１０】図２はターゲットシステムの一実施例として好適なパーソナルコンピュータ２００の構成図である。

【００１１】１．ソースコードのコンパイル

１－１．ターゲットシステムの説明

本実施例のターゲットシステムであるパーソナルコンピ

ュータ２００は．制御装置２０４によって、表示装置、入力装置等のユーザーインターフェース装置を制御し、使用者の操作を受けつける。プロセッサ１１３は汎用のマイクロコンピュータであり、オペレーティングシステム２０３の処理を実行し．プロセッサ資源、メモリ資源の管理を行なっている。中間言語インタープリタ１１１はこのオペレーティングシステム２０３の機能を呼び出して実行される独立したプロセスである。プロセスはオペレーティングシステムにおけるプロセッサ資源．メモリ資源割り当ての実行時の単位である。本実施例でプロセスは、プロセスの識別子及び実行管理、メモリ管理のための情報を含むプロセスヘッダと．中断の際に現在のプロセッサのレジスタの状態を保存するための領域と、オブジェクトコード領域．スタック領域からなる。パーソナルコンピュータ２００において多重処理を行なう場合は．複数のプロセスを起動し、プロセスに対するスケジュールによって処理を多重化する。また、中間言語で記述された目的プログラムのオブジェクトコードは、それぞれ中間言語インタープリタ１１１を起動してこのインタープリタ上で実行される。

【００１２】パーソナルコンピュータ２００のプロセッサ１１３に対して使用者は．ハードウェア付加によって機能向上を図ることができる。パーソナルコンピュータ２００はこの目的に応えるため外部拡張用のシステムバス２０２を持つ．図２の構成ではシステムバス２０２を使用して付加ハードウェア２０１を拡張した様子を示した。ここで、プロセッサ１１４はプロセッサ１１３の処理の部分を分担し、後述する方法で並列実行する事によってパーソナルコンピュータ２００の処理速度を向上させる。この様な付加ハードウェアはその役割からアクセラレータと呼ばれる事がある。プロセッサ１１４はシステムバス２０２を介してのデータの授受、割り込み等を処理するためカーネルプログラム２０５を実行する。中間言語インタープリタ１１２はこのカーネル２０５に管理されるプロセスである．プロセッサ１１４がこれらプログラム実行をプロセッサ１１３とは独立して行なうため．局所メモリ２０６が使用される。また、プロセッサ１１４とシステムバス２０２の接続は先入れデータを先に取り出すことが出来るように構成された順序付きメモリであるＦＩＦＯ２０７．２０８によって行なわれる。ＦＩＦＯ２０７．２０８は．プロセッサ１１３のメモリ空間にアドレス割り当てされプロセッサ１１３からアクセスされる。

【００１３】１－２．並列実行及び同期、排他制御の記述方法

本実施例では．複数プロセッサによる並列実行をプログラム言語記述の段階でソースコード中に明示的に記述する。このために並列記述を認める言語仕様が必要である。ここでは言語Pascalに並列記述のための述語を追加した言語を取り上げて説明する。本実施例では説明の簡略化のため言語Pascalの文法の中でprocedure文またはfunction文で開始される副プログラムをブロックと呼び．代入文とif．while．repeat．for等の出発記号で開始される文をステートメントと呼ぶ。またbegin...endで囲まれた「文」の並びは．通常は「複合文（compound statement）」と呼ぶが．ここではこれもステートメントと呼ぶ。追加した仕様は次の２種類である。

【００１４】（１）cobegin,coend

cobegin,coendはそれぞれ並列実行を認めるブロックの開始文および終了文である。

【００１５】（２）モニタ(monitor)型

モニタ（monitor）型は、並列実行される複数のブロックによってアクセスされるいわゆる共有変数を宣言する型付け（typing）である。モニタ型は共有資源を抽象化して表す形式と見なす事もできるが．プログラム上は特定された手続きによってだけアクセスできる資源を与える。この方法で共有変数に対する排他制御及び同期を取り扱う方法は複数の公知例に詳しい（例えば、上田和紀著：並列プログラミング言語、情報処理、Vol.27．No.9．pp.995-1004(1986)、Brinch Hansen,P.著：The Programming Language Concurrent Pascal．IEEE Trans．Software Eng.、 Vol.1．No.2、pp.199-207(1975)など）。

【００１６】図３は、モニタを用いたプログラムの説明図である。プログラム３００は、手続き３０５が何らかの整数型のデータを生成し．手続き３０６がそれを次々に消費していくという処理内容を並列記述言語によって記述した例である。これは例えば、手続き３０５が外部装置からデータを受信する処理で、手続き３０６がそのデータを加工して表示する場合などに相当する。ここで変数bufをモニタ型であるbuffertypeとして宣言した事により、メモリ上の共有資源の実態である変数（変数名はbuffer）３０８にはモニタ型の宣言３０１内部で定義した手続き３０２または３０３によってしかアクセスする事ができない。またモニタ内の各変数は手続き３０４で初期化される。生産者手続き３０５が手続きbuf.append(v)によって変数３０８への代入を行なおうとすると．実際には処理３０２が実行される。この時．既に配列変数３０８であるbufferの中味が全て満たされているとすると、この要求は待ち行列putqを作って待たされる（３０９）（これを手続きwait(putq)として表示した）。それ以外の場合は配列変数３０８への要素の追加が行なわれ、ポインタが更新された後．手続きsignal(qetq)が呼び出される（３１０）。手続きsignal(qetq)は．待ち行列qetqを作って待たされているデータ取得の処理呼び出しが有ればそれを実行し．結果を返す処理を行なう。消費者手続き３０６が手続きbuf.fetch(v)によって変数３０８の内容の１要素を読み取ろうとすると、処理手続き３０３が実行される。この時、配列変数３０８であるbufferの内容が空であれば．この要求は待ち行列qetqを作って待たされる（処理３１１の手続きwait(q

etq)）。それ以外の場合は配列変数３０８の内容が読み出され、ポインタの更新が行なわれた後、手続きsignal(putq)が呼び出される（３１２）。手続きsignal(putq)は上記と同様に待ち行列putqに手続き呼び出しが有ればそれを実行し、結果を返す。モニタ型の構造はモニタ型のデータが宣言されたブロックの管理下にある資源と見なされる。本実施例ではこの資源へのアクセスは中間言語インタープリタによって行なわれる。インタープリタの処理自体は逐次的な処理であるから、結果的に共有変数への排他制御および処理同期が保証できる。モニタ型定義を検出した場合の中間言語発生方法は次に述べる。

【００１７】１－３．字句解析、構文解析の処理

ソースコード１０１を読み取ったコンパイラ１０２は、字句解析手段１０３により字句解析を行なう。ここでは、ソースコード１０１の中に現われる文字列を順次読み取ることによって手続き、変数、定数、文字定数、データ型定義、文字列等の名前と値、さらに演算子、特殊記号、プログラム言語の予約語の取り出しを行なう。この読み取り結果はコンパイラ１０２の作業領域内に記録される。

【００１８】次に、字句解析手段１０３の読み取り結果のデータを入力として構文解析手段１０４が実施される。本実施例の構文解析符は言語Pascalの文法がLL(1)と呼ばれる性質の文法であるため、周知の下降解析を行なう（下降解析に関する公知例には中田育男著：コンパイラ、産業図書(1981)等多数がある）。言語Pascal等のブロック構造を持つ言語においては、変数名、ブロック＊名は宣言されたブロック内を有効範囲とする。このため、変数の値の決定、参照にはブロックの深さと（ブロックの深さが同じ場合は）ブロックの登場順序を表す情報が必要である。構文解析手段１０４ではプログラム開始のレベルをブロックレベル＝０とみなして処理を進める。この後、ブロックの入れ子の状態が１段深くなる毎にブロックレベルの値は＋１される。同じく、入れ子の構造を１段抜け出す度に、ブロックレベルの値は－１される。すなわち主プログラムがブロックレベル＝０であり、主プログラム内で宣言される手続きがブロックレベル＝１である。

```
procedure block(....);
  var
     ident : 述語;    ...;
  begin
    (* 次の述語を取り出しidentに代入 *)
    ident := next_word_in_source;
      if (ident = 'procedure')
          or (ident = 'function') then block(...) (*再帰*)
      else
      begin
          ....その他の構文...;
      end
  end;
```

この繰り返しを正しく制御するため手続きまたは関数の宣言を検出しブロックに入ると（Ｓ４０１）、ブロックレベルを＋１し（Ｓ４０２）、ブロック番号を＋１する（Ｓ４０３）。この一方で、ブロックの終了を検出した場合は、ブロックレベルを－１し（Ｓ４２３）、その結果ブロックレベルの大きさが０より小さいか調べる（Ｓ４２４）。ブロックレベルが０より小さくなった場合は、主プログラムのレベルで、ブロックの終了を検出した事になり、プログラムの構文解析が完了した事を示している。また、ブロックレベルが０以上であればさらに構文は続く可能性があり、処理を継続する。以上が構文解析手段１０４の処理の流れの概要である。次に個々の処理ついてさらに説明する。

【００１９】本実施例の構文解析手段１０４が従来方法と異なる部分は、その内部で共有メモリ資源検出手段１０５と、並列実行検出手段１０６を処理する点である。前者は、並列実行される複数の処理単位によってアクセスされる変数を定義し、その変数への処理手順を確立する実行コードを生成する。後者は、並列に実行される手続き、関数呼び出しのためのコードを生成する。

【００２０】構文解析手段１０４の処理手順を図４の流れ図を用いて説明する。ブロック構造を持つ言語の構文解析を行なう事から、本実施例の構文解析手段１０４は、再帰的な下降解析を行なう。すなわち、ブロック検出とブロック内での構文解析を行ない、ブロック内に更にブロックの定義があれば、自分自身の処理を再帰的に呼び出し、ブロック検出を行なう。これは、疑似的なプログラム言語で次の様に書くことができる。

【００２１】

【００２２】構文解析の処理では、個々の処理ブロックについて実行コードの開始番地、変数表の格納位置、引数渡しに使用するスタック領域の大きさ等の情報を記録する必要がある。本実施例はこの目的に実行時ブロック管理テーブル５０１とブロック番号管理テーブル８０１を使用する。図５は本実施例で使用する実行時ブロック管理テーブル５０１のデータ構造を説明する図である。

実行時ブロック管理テーブル５０１は、データ構造５０２を一つの要素とする配列の形式をとる。データ構造５０２は、ブロック番号５０３、下位ブロックへの連鎖５０４、オブジェクトコード上でのオフセット５０５、引数渡しに消費するスタックの深さ５０６、引数リストへの連鎖アドレス５０７によって構成される。

【００２３】再び流れ図に戻り処理手順の説明を続ける。

【００２４】処理Ｓ４０３によって、ソースコード中の全ての処理ブロック（手続き、関数）はユニークなブロック番号が与えられる。各処理ブロックのブロック番号とブロックレベルの関係を保持するため、本実施例の処理では図８ａに示すブロック番号管理テーブル８０１が生成される。ブロック番号管理テーブルの１つの要素は、ブロック番号８０２、ブロックレベル８０３、テーブルエントリ８０４からなるデータ構造である。処理Ｓ４０３では、この内ブロック番号８０２、ブロックレベル８０３の値の記録が行われる。また処理Ｓ４０３は、現在の処理ブロックのコード生成の開始位置が全プログラムのオブジェクトコード開始位置から相対位置で何バイトの位置であるか計算を行う。この値は実行時ブロック管理テーブルのデータ構造５０５に記録される。

【００２５】個々のブロック（手続きか関数）はその呼び出し時に引数を受け取ることができる。このとき、中間言語インタープリタでは、スタック領域として確保したメモリにその引数を積み、処理呼び出しを行なう。そこで並列実行時にある手続きを別のプロセッサにスケジュールして実行するためには、このスタック領域に配置される引数を解釈し、別プロセッサの中間言語インタープリタの作業領域に複写する必要がある。このため、引数リスト解析処理（Ｓ４０４）が行なわれる。この処理は、引数リストを実行時ブロック管理テーブル５０１のデータ構造５０７に連結リストとして記録する。この処理の詳細は次の節で説明する。

【００２６】続いてブロック間のリンク発生を行なう（Ｓ４０５）。これは並列実行時の処理ブロック転送に必要な情報を生成するものである。詳細は後述する。

【００２７】この後処理は定数宣言を検出した場合（Ｓ４０６）はそれを名前表に登録し（Ｓ４０７）、型宣言を検出した場合（Ｓ４０８）は、型宣言辞書への登録を行なう（Ｓ４０９）。通常、構文解析処理ではソースプログラム中に現われる予約語以外の名前は定数名、変数名、手続き名、関数名、変数型名を区別せず名前表に登録し、この表中に名前と共にそのオブジェクトタイプを記録する方法を採る場合が多い。これは名前の多重定義（関数名と定数名が同一など）の不都合を未然に検出する上で合理的な方法である。本実施例も構文解析手段１０４における名前（識別子）の管理はこの方法によった。

【００２８】但し変数宣言を検出した時（Ｓ４１０）、

共有メモリ資源検出手段１０５においてモニタ型変数が検出された場合（Ｓ４１１）は、通常の名前表への登録の他に後述するような共有変数処理（Ｓ４１２）が実施される。モニタ型以外の変数であればブロック内の変数についての表作成が行われ（Ｓ４１３）、さらに、局所変数割り当てのコード生成が処理される（Ｓ４１４）。ここで、処理Ｓ４１３でブロック内の変数について作成された変数表のコンパイラ１０２の作業領域上での開始番地がブロック番号管理テーブル８０１中のテーブルエントリ８０４の値として記録される。

【００２９】次にブロック開始以外の構文についての処理の流れを説明する。

【００３０】本実施例では言語仕様に並列記述を認めているため、並列実行すべき部分はソースコード中に明示的にに記述することを要求している。言い換えるとcobegin、coendで囲まれた範囲に含まれない処理ブロックは逐次処理環境での実行コードが生成される。これを構文解析手段１０４、コード生成手段１０７、１０８、１１７の立場から見れば、cobegin、coendで並列実行を指定された処理単位以外のブロックの実行コードは通常のコンパイル処理時と同じコード生成を行うことになる。そこで構文解析手段１０４は並列実行検出手段１０６を実施する。その処理はソースコードの構文中に予約語cobeginを検出すると並列実行フラグ＝［真］と設定し（Ｓ４２０）、予約語coendを検出した場合は並列実行フラグ＝［偽］とする（Ｓ４２１）処理である。実際の実行コード（中間言語文）の生成はこのフラグの真偽を判断しながら行われる。

【００３１】取り出された述語が識別子であってしかも名前表登録を検索した結果、手続き名あるいは関数名であったとすると、他の処理ブロックを呼び出す処理であると判断できる（Ｓ４１５）。この判断が［真］であるときはまずブロック間のリンク処理が行われ（Ｓ４１６）、次にブロック呼び出しのコード生成が行われる（Ｓ４２５）。この時、上述した「並列実行フラグ」の内容が検査され、フラグ＝［真］の場合はコード生成手段１０８が実行される。その他の場合はコード生成手段１１７が実行される。

【００３２】また、取り出した述語が「モニタ型変数名」＋「．」＋「手続き名」という名前の構造を持つ時はモニタ型変数名によって名前表を検索し、名前が存在した場合モニタ型手続き呼び出しであると判断できる（Ｓ４１７）。この場合は共有資源アクセス手続き生成処理（Ｓ４１８）が実行される。この処理の下部構造としてさらにコード生成手段１０７が処理され、中間言語文が成生される。

【００３３】以上が構文解析手段１０４の処理の概要である。次に、各部分の動作について順次説明する。

【００３４】１－３－１．引数リスト解析処理（Ｓ４０４）

周知の様にPascal系の言語では手続きに引数が渡される場合．引数参照の形式には２通りある．「値を渡す引数（call by value）」と「名前を渡す引数（call by reference）」である。本実施例では、引数参照型の内部表現として値渡しの場合" 参照型＝０" ．名前を渡す場合" 参照型＝１" とした。

【００３５】引数の並びは構文の中ではブロック名に続く" （" で開始され、変数名と変数のデータ型（整数、配列．文字等）の対で表記される。また引数並びの終了は" ）；" である。さらに「名前を渡す引数」は．予約語" var" で開始される。引数リスト解析処理Ｓ４０４は．予約語" var" を検出した場合、参照型＝１とし．それ以外の場合は参照型＝０とする。次に変数のデータ型を解析し、そのデータ型に与えられた内部表現の値を取り出す。使用可能なデータ型には全て内部表現としてユニークな値（整数値）が与えられている。解析処理Ｓ４０４は、この" 参照型．変数型" の値を実行時ブロック管理テーブルの連鎖アドレス５０７から続くデータ構造５０９のリストに追加する。データ構造５０９は、連結リストのデータ型をとり．データの末尾は連鎖アドレスである。連鎖アドレス５０７からの追鎖５０９は、引数リストとしてコード生成処理時に参照される。

【００３６】１－３－２．ブロック間の従属関係の抽出（Ｓ４０５、Ｓ４１６）

本実施例の処理系では、並列実行される対象のブロックは（それが可能であれば）別のプロセッサの局所メモリ空間に転送され実行される。このためには、ある処理ブロックを他のプロセッサに配置した時．この処理ブロックから呼び出される手続き．関数のすべてを取り出し、この複写を前記の処理ブロックと同じプロセッサの局所メモリに配置することが望まれる。これはある処理ブロックが下位ブロックを呼び出す度にプロセス間通信する必要を除くための処理である。この実現には、並列実行の対象となるある処理ブロックに注目した時、その処理ブロックが呼び出す（以下これを従属すると言く）ブロックを特定できる様な管理手段が実施されれば良い。本実施例は各処理ブロック間の従属関係を保持するデータ構造を実行時ブロック管理テーブル５０１に保持することでこの管理手段を実現した。

【００３７】図４の流れ図中の処理Ｓ４０５、Ｓ４１６はこの実行時ブロック管理テーブル中に、実際のブロック間のリンクを記述する処理である。

【００３８】本実施例の言語の特性から図６のソースコードの様にプログラムが記述された時．処理単位間のブロック構造と構文解析時に処理Ｓ４０３で与えたブロック番号を図示すると図７の様に示すことができる。変数名．手続き名など識別子に与えられたスコープ（有効範囲）の考え方に従うと、例えばブロック番号１１である手続きp321の処理内部では手続きp32，p3など上位構造の変数にアクセス可能であるが、手続きp31，p2等の変数へのアクセスは許されない。

【００３９】この様な変数スコープの関係を正しく維持しながら、処理ブロックの階層関係を取り出すため処理Ｓ４０５はブロック番号管理テーブル８０１を参照しながら処理を行う。この処理を図９の流れ図に示した。自ブロックのブロックレベルが０、１のどちらかであれば．自ブロックはトップレベルで宣言されており．上位階層へ従属しないためリンク発生が不要である（Ｓ９０１）。これ以外の場合は．ブロック番号管理テーブル８０１の自ブロックの位置からテーブル内の要素をさかのぼり．ブロックレベルの値が自ブロックのブロックレベルの値より一つ小さな値を持つ要素を捜す（Ｓ９０２）。その要素のブロック番号を取り出し（Ｓ９０３）．そのブロック番号の値をキーとして実行時ブロック管理テーブル５０１の中から一致するブロック番号５０３を持つブロックの位置を見つけ．その要素へアクセスする。この要素の下位ブロックへの追鎖アドレス５０４からの連鎖５０８を生成する（Ｓ９０４）。以上の処理によって上位構造に位置するブロックについて自ブロックへの連鎖を記録することができる。図８ｂは図６のプログラムを例に採ってブロック間のリンク動作を説明した図である。解りやすくするため．ブロック番号管理テーブル８０１の内容の右側にブロック間のリンクの様子を矢印で示した。図６のプログラムから図７のようにブロック構造に従ってブロック番号を与えた時．例えばブロック番号＝１０の手続きp32はブロック番号＝５の手続きp3に従属する。図８ｂに示すように手続きp32のブロックレベルの値は２であるから．ブロック番号管理テーブル８０１をブロックレベル＝１であるような要素を捜してさかのぼるとブロック番号＝５の手続きp3の位置を検出できる。

【００４０】他方、処理Ｓ４１６は自ブロックが呼び出すブロックへの追鎖を生成する処理である。この処理はより単純である。すなわち．構文中の識別子に従って名前表を検索した結果、その名前が手続きあるいは関数の名前であったら．そのブロック番号を取り出す。このブロック番号の値を実行時ブロック管理テーブルの自ブロックの位置の追鎖アドレス５０４に続く連鎖５０８に書き加えるだけである。また．この連鎖５０８の生成にあたり既に同一ブロック番号の要素が追鎖上にあれば重複を避けるため追鎖への追加は行わない。

【００４１】１－３－３．共有変数処理（Ｓ４１２）

共有変数処理Ｓ４１２は実際のモニタ型変数へのアクセスを伴う実行コードの領域を確保する。次にこの領域に中間言語インタープリタの標準手続きとして用意されたモニタ型手続きの目的コードをロードする。この時、個々のモニタ型変数にはソースコード内で宣言された順にユニークな番号が与えられる。これは図１０に示したデータ構造を持つ共有資源管理テーブル１００１に記録される。共有資源管理テーブル１００１の一つの要素は、

資源番号１００２、共有変数名１００３、共有資源を宣言した処理ブロックのブロック番号１００４、モニタ型手続きの目的コード１００６へのオフセットの値１００５から構成される。

【００４２】２．中間言語によるコード生成

次に、中間言語の生成手順について説明する。説明の簡単のため中間言語についてその機能を説明する表の一部を図１１に図示した。但しTOPは中間言語インタープリタの作業領域に取ったスタック型データ領域の最上段の番地を表し、SPはスタック型データ領域を指し示すポインタを表す。本実施例のインタープリタスタックはアドレス値の小さい方に新たに要素を積み重ね、要素を取り出す時はアドレス値の大きな方向にポインタを更新する先入れ後出し型のメモリ構造である。また識別子xの格納されるアドレスをaddr(x)と書き表す。

【００４３】中間言語インタープリタは、

LODV addr(x)

と書かれた命令をアドレスaddr(x)に格納されている値を取り出し、それを中間言語インタープリタの実行時のスタックの最上段に格納せよという命令として解釈する。（以下の説明では記述の簡単のため、これを単に" LODV x "と書き表す。）また、

LODA addr(x)

という命令はaddr(x)のアドレスの値そのものを評価し、スタックの最上段に置く。

【００４４】LODN obj

はobjが処理ブロックまたはモニタ型の共有資源であり、オブジェクトのブロック番号か資源番号の値を取り出し、スタックの最上段に置く。

【００４５】LOD n

はインタープリタの使用するスタック中で現在のスタックポインタから+nバイトの深さのアドレスを計算しそのアドレスにあるデータを取り出し、スタックの最上段に複写する。

【００４６】ALOC n

は現在のスタックポインタを番地の少ない方向に向かってn進め（つまりnだけ減算し）、局所変数のための領域を割り当てる。

【００４７】UALC n

はALOC nと対をなす命令で、スタックポインタの値にnを加えALOC n 命令で空けた領域を元に戻す。

【００４８】CALL addr(x)

は識別子Xの先頭番地へ処理分岐し、命令語RETを検出した場合、先にCALL Xを実行した次の命令位置に復帰する。

【００４９】ADD

はスタック最上段の値と、スタックの次の位置にある値を加え、スタックポインタを１段スタックの減る方向に進めてスタックの新たな最上段に加算結果を格納する。

【００５０】中間言語はこのほか演算処理、条件分岐処理等の命令語を持つ。いずれの命令もプロセッサのレジスタをアキュムレータとして使用せず、スタック上で演算操作を行う形式とした。言うまでもなくこの中間言語の仕様は並列実行する中間言語のインタープリタ間で統一されてさえいれば他の仕様であっても構わない。また、ここで使用するスタックは中間言語インタープリタの作業領域に取ったリニアなメモリ空間でプロセッサスタックと異なる。

【００５１】図１３は中間言語記述生成の処理の一部を疑似的なプログラム言語によって記述した説明図である。記述はブロック呼び出し時のコード生成手順１３０１と、cobegin、coend文を検出した場合のコード生成手順１３０３を示している。ブロック呼び出し時の処理には引数に対するコード生成手順１３０２が含まれる。この他に算術演算の際の中間言語生成が必要であるが、従来方法と同じであるため省略した。中間言語生成の処理はある条件を検出したとき、その条件に適用されるルールに従い中間言語発生する方法をとる。従って処理は条件の検出とルール化されたコード生成に対し、構文解析で取り出した識別子、値、番地等の評価を組み込む処理を繰り返す操作である。

【００５２】以下に例を挙げて上記中間言語を実行コード中でどのように発生するかを説明する。

【００５３】２－１．並列実行時のコード生成手段１０８

中間言語による記述の生成を図１２を用いて説明する。ソースコード中に１２００に示す手続きp1の定義が現われた場合、構文解析手段１０４は図４の流れ図で示した様に引数リスト解析を行う（Ｓ４０４）。第1引数１２０１は識別子xで始まる。構文解析手段１０４はこれを「値を渡す引数」であると判断する。この結果、参照型＝0が記録され、次に引数の型がintegerであることから整数型の内部表現である1が" 型名 "として記録される。第2引数１２０２は予約語varで始まる。ゆえに構文解析手段１０４はこれを「名前を渡す引数」と判断し、" 参照型＝1 "、" 型名＝1 "が記録される。この結果、実行時ブロック管理テーブル５０１の連鎖アドレス５０７からの連鎖５０９によって引数リストが形成される。手続きp1に関する引数リストの内容を取り出すと１２０３のように図示できる。

【００５４】ソースコード中に記述１２０４が現われた場合、並列実行されない場合はコード生成手段１１７が実施され、引数リスト１２０３の内容に従い、中間言語記述１２０５が生成される。この記述の生成は引数リスト１２０３に現われる順にルール化された命令語を配置する方法によったもので周知の方法である。

【００５５】本実施例では並列実行する部分について、従来方法と異なる中間言語記述の発生を行う。前提条件として本実施例では並列実行が許される手続き、関数はモニタ型変数以外の変数引数を認めない。これは変数引

数が代入を認める引数であることを考えれば当然である。手続きp2を第1引数に値渡しの整数、第2引数に名前渡しのモニタ型変数を持つ手続きとし、手続きp3を引数を持たない手続きとする。ソースコード中に記述1206が現われると本実施例の構文解析手段104は次の手順の処理を実行する。

【0056】手順1　並列実行検出手段106がcobegin文を検出し並列実行フラグを［真］とする（図4の流れ図では処理S420）。

【0057】手順2　構文解析手段104の中で記述"p2(a,b);"が解析され、ブロック呼び出しであると検出される（S415）。

【0058】手順3　並列実行フラグの内容が検査される。ここでは［真］であるためコード生成手段108が実行される。

【0059】手順4　コード生成手段108によってコンパイラ102の作業領域の名前表が検索され、手続きp2のブロック番号と引数リストが指定される。

【0060】手順5　引数リストの内容から、第1引数が値渡し（参照型＝0）であり、整数型であると知ることができる。

【0061】手順6　コード生成手段108が生成ルールに従い中間言語の述語を出力する。ここでのルールは「値渡しの引数は、命令語｛LODN x｝形式を出力」である。

【0062】手順7　引数リストの内容から、第2引数が名前渡しの変数で、モニタ型であることを知ることができる。

【0063】手順8　ルールとして「モニタ型変数引数は、命令語｛LODN obj｝形式を出力」が用いられ、中間言語記述が出力される。

【0064】手順9　手続きp2を呼び出す中間言語記述を発生する。ここでは並列実行時のルール「手続き呼び出しは命令語｛LODN obj, CALL addr(coexec)｝形式を出力」が適用される。

【0065】以下、同様に手続きp3についても処理が行われる。さらに次の手順が処理される。

【0066】手順10　並列実行検出手段106がcobend文を検出し並列実行フラグを［偽］とする（図4の流れ図では処理S421）。

【0067】手順11　並列実行終了時のルールに従い、命令語｛CALL addr(sync)｝が出力される。

【0068】以上の手順の結果として記述1207が出力される。ここで手続きcoexecとsyncは中間言語インタープリタの組み込み手続きである。これら組み込み手続きの動作は"3－3．並列記述部分の実行時処理"の節で説明する。

【0069】2－2．共有資源アクセス手続きのコード生成手段（S418）構文解析手段104において、共有資源へのアクセスを伴う構文が検出された時、処理S418が実行される。共有資源へのアクセスを伴う中間言語の発生は疑似的なプログラム言語で書き表すと、図13の1304のように示すことができる。モニタ型変数へのアクセスを行うためには、モニタ型変数が非局所変数として手続き中に指定される場合と、引数リスト中に指定される場合がある。

【0070】前者のコード生成は単純である。コードが実行された場合を図14aを使って示すと、中間言語インタープリタはスタック上にモニタ型手続きへの引数1404を置き、さらに上段にモニタ型資源番号の値1405を置いてappend, fetchの手続き呼び出しを行う。これはコード生成側から見ると次の手順による中間言語発生を行うことで実現できる。

【0071】手順1　構文解析手段104の処理S418が構文中に記述されたモニタ型の名前をキーとして共有資源管理テーブル1001を検索し、資源番号を取り出す。手順2　コード生成手段107が呼び出される。

【0072】手順3　プログラム1304で記述した手順が実行される。直前までの目的コード生成によりモニタ型手続きに対する引数は、スタック最上段1404に置かれる。これに続いて命令語｛LODN obj｝を出力する。

【0073】手順4　構文解析結果がappend呼び出しかfetch呼び出しか判断し、命令語｛CALL addr(appendまたはfetch)｝を出力する。

【0074】一方、複数の共有資源が有り、同一手続きを使用しながらもアクセスする資源を場合によって区別したいという要求の下では、モニタ型引数が必要となる。この場合のコード生成はやや複雑である。モニタ型引数をさむコードとはソースコード中に記述1208が現われた場合等である。図14bの実行時のスタックの使用の履歴1401を用いて説明する。

【0075】手続き呼び出し時の引数は、スタック上に積み重ねられる。この後、実際に手続きが呼び出されると手続き内の局所的な変数を確保するため命令｛ALOC N｝が実行され、局所領域1402が確保される。さらに消費等の目的コードがスタックを消費し、現在の処理結果がこの時点でのスタックの最上段1404に配置される。これがモニタ型手続きへの引数である。次にモニタ型の資源番号をスタックの最上段に置く。このために命令語｛LOD +d｝が実行され、スタックの＋dバイトの深さの位置から矢印1403の様に値をスタックトップに複写する。続いてappend, fetchの手続き呼び出しが実行される。これはコード生成側から見ると、次の手順による中間言語発生を行うことで実現できる。

【0076】手順1　局所変数のための領域の展開を行う。このために命令語｛ALOC N｝を出力する。

【0077】手順2　他の目的コード生成をする。

【0078】手順3　モニタ型手続き呼び出しで検出したら引数リスト中のモニタ型変数がスタックに置かれた

位置までの深さを計算する。計算結果をdとして命令語{LOD+d}を発生する。

【0079】手順4　構文解析結果がappend呼び出しかfetch呼び出しか判断し、命令語{CALL addr(appendまたはfetch)}を出力する。

【0080】手順5　もし必要ならさらに他の目的コード生成をする。その後命令語{UALC N,RET}の順に出力する。

【0081】以上の処理手順の結果としてソースコードの記述1208から中間言語による目的コード1209が出力される。

【0082】2－3．オブジェクトファイル作成手段109の動作

上述した手順によって目的コード記述が得られる。オブジェクトファイル生成手段はコード生成手段107、108、117により生成される目的コード記述を入力として中間言語オブジェクト110を出力する。目的コード記述は、手続き呼び出しを手続きの名前（識別子）に結び付けられたブロック番号で記述している。これは実質的に名前と等価であるから上記説明では手続き名を用いて説明した。しかし、実行時に必要なのは手続きの名前ではなく、その手続きがプログラム中で先頭位置から相対値で何バイトの位置に記録されているかである。この情報は既に説明した実行時ブロック管理テーブル501のデータ構造505によって保持されている。そこでオブジェクトファイル作成手段109は、目的コード記述を読み取り、ブロック名の呼び出しを検出するとこのブロック名（正確にはブロック番号）をキーとして実行時ブロック管理テーブル501を検索し、実行コードでの相対位置505を取り出す。次にこの値によって中間言語記述のブロック名を相対位置505の値に置き換える。

【0083】同様に共有資源であるモニタ型変数／手続きが目的コード記述では資源番号で指定されている。オブジェクトファイル作成手段109はこれについても資源番号をキーとして共有資源管理テーブル1001を検索し、実行コード上でのオフセットの値1005を取り出し、目的コード記述の中で資源番号として記述されている部分を置き換える。

【0084】以上の処理に続き、オブジェクトファイル作成手段109は実行時ブロック管理テーブル501と共有資源管理テーブル1001の内容を目的コード記述の後に書き加え、中間言語オブジェクト110としてファイル出力する。

【0085】3．中間言語インタープリタ

3－1．中間言語インタープリタ間の通信

上述したコンパイラ処理系はターゲットシステム上で実行される必要はない。他のコンピュータシステムをホスト装置として中間言語オブジェクト110を生成することが可能である。生成された中間言語オブジェクトはファイル出力され、ターゲットシステムには磁気ディスク媒体、半導体ROM、通信手段等によって伝達される。あるいはターゲットシステムで直接上述の構成のコンパイラを実行し、ソースコードから中間言語オブジェクト110を生成することも可能である。

【0086】中間言語インタープリタは、この中間言語オブジェクト110を入力として動作する処理系である。中間言語インタープリタは与えられた中間言語を解釈し実行できる事が必要条件であり、このためにどのような実現形態でも良い。もっとも単純な実施例は図16の流れ図に示す様な無限ループを続ける装置を用いる方法である。本実施例は図2で示すようにプロセッサ113が実行するオペレーティングシステム203の上にプロセス111を起動し、このプロセス111によって図16の流れ図の処理を実現した。また、プロセッサ114では割り込み処理等をサポートするカーネル205の上にプロセス112を起動し、同様の処理の流れを実現した。従ってここではプロセス111、112を中間言語インタープリタ（以下IPRと略す）と呼ぶ。また説明のため主プログラムの起動が行われたプロセッサをローカルプロセッサと呼ぶ。また、並列実行記述を検出した後、一部の処理ブロックが分散され並列実行を担うプロセッサをリモートプロセッサと呼ぶ。同様に主プログラムの起動が行われたコンピュータ上の中間言語インタープリタをマスタIPRと呼び、リモートプロセッサ上で実行される中間言語インタープリタをスレブIPRと呼ぶ。

【0087】IPRとして動作するプロセスは、図17のプロセス1702として表す構成をとる。プロセス1702はプロセス管理のための情報が記録されるプロセスヘッダと、プロセッサプログラムカウンタ1703が指し示す実行コード領域と、作業領域と、プロセッサスタックポインタ1710が指し示すプロセッサスタック1711からなる。実行コード領域には、インタープリタプログラムのオブジェクトコード1704と、組み込み手続き群のオブジェクトコード1705が格納される。中間言語オブジェクトコードが配置されるコード領域1706と、中間言語用のスタック領域1707は、プロセスの作業領域に配置される。また同じ作業領域には中間言語用の命令ポインタ1708と、中間言語用のスタックポインタ1709が置かれ、この他、実行時ブロック管理テーブル501、共有資源管理テーブル1001、実行時プロセッサ管理テーブル1701が置かれる。IPR111、112はFIFO207、208を使って通信しあう事ができる。FIFO208にデータ書き込みがあるとシステムバス202を介してプロセッサ113に割り込みが発生する。この割り込みはオペレーティングシステム203の処理ルーチンに取り出され、IPR111に伝達される。またFIFO207にはシステムバス202によりプロセッサ113のアドレス空間における

アドレス割り当てが行われている。このためプロセッサ１１３がこのアドレスにデータ書き込みを行うとFIFO２０７はプロセッサ１１４への割り込み信号を発生することができる。プロセッサ１１４のカーネル２０５は割り込みサービスルーチンによってFIFO２０７に書き込まれたデータを取り出し、ｉＰＲ１１２に伝達する。

【００８８】ＩＰＲ１１１．１１２はこのFIFO２０７、２０８を用いて通信しあうプロセスと見なすことができる。本処理系の設計上、パーソナルコンピュータ２００に対し付加されるハードウェアは複数個許容できる。このため、マスタＩＰＲ（この場合はＩＰＲ１１１）とスレブＩＰＲの通信は１対１の通信に限定されない。そこで、本実施例ではｉＰＲとして動作するプロセス間の通信に図１５に示す１５０１．１５０５または１５１０の形式のデータ構造を持つデータ列を用いる。このデータ構造において、プロセッサ番号１５０３はシステムで一意に決定する各プロセッサの番号であり、プロセス番号１５０４は各プロセッサがプロセス生成時にプロセスに割り当てた番号である。従って、複数のプロセスが存在してもプロセッサ番号１５０３とプロセス番号１５０４の両者を指定する事であるプロセスを特定できる。一方、コマンド１５０２として使用されるのは｛コネクト・ロック｝、｛ロード｝、｛実行｝、｛リリース｝、｛フェッチ｝、｛アペンド｝の６つである。通信１５０１．１５０５を受信したＩＰＲは応答として１５１０のデータ構造からなるメッセージを返す。ここで状態コード１５１１として｛ビジー｝、｛レディ｝、｛アクセプト｝、｛ダーン｝、｛フェイル｝の５つが使用される。

【００８９】以下図１６の流れ図に従ってｉＰＲの動作を説明する。ＩＰＲはコマンドを受信すると（Ｓ１６０１）その内容が｛コネクト・ロック｝か判断する（Ｓ１６０２）。｛コネクト・ロック｝であればｉＰＲ内部の状態コードを｛ビジー｝とし（Ｓ１６０３）コネクト処理Ｓ１６０４を実行する。コネクト処理Ｓ１６０４は受信したデータ列からプロセッサ番号１５０３とプロセス番号１５０４を取り出し、内部作業領域に記録し、｛コネクト・ロック｝処理を発生したプロセスに対し状態コード｛アクセプト｝を返す。これ以降はコマンドを受け取った後、これを送信したプロセスがコネクト処理の対象となったプロセスであるか判断し（Ｓ１６０５）、Ｎｏであれば状態コード｛ビジー｝を返す（Ｓ１６０６）。コネクト処理の対象となったプロセスからのコマンドを受け取った場合は、｛ロード｝、｛リリース｝、｛実行｝のいずれかのコマンドを実行する。｛アペンド｝、｛フェッチ｝コマンドは、コネクトされたプロセス以外のプロセスに対しても応答される。

【００９０】コマンドが｛リリース｝の場合は、状態コードを｛レディ｝とし（Ｓ１６０７）、コネクト処理のため記録したプロセッサ番号、プロセス番号を廃棄し、コマンド読み取り処理Ｓ１６０１に戻る。

【００９１】コマンドが｛ロード｝であればロード処理Ｓ１６０８を実行する。ロード処理Ｓ１６０８はスレブＩＰＲの作業領域に中間言語記述されたオブジェクトコード１５０７を読み込み、次にスタック初期化データ１５０８を読み込み、このデータの指示に従いスタックに初期状態のデータを格納しスタックポインタを設定する。続いて、テーブル初期化データ１５０９を読み込み、実行時ブロック管理テーブル５０１、共有資源管理テーブル１００１の内容を初期設定する。

【００９２】コマンドが｛実行｝の時は、実行状態に入り、コード領域１７０６の命令ポインタの指定する命令から、逐次実行処理（Ｓ１６０９）に入る。逐次実行処理Ｓ１６０９は命令語処理群１６１０を実行するほか、処理内容に応じて組み込み手続き処理群１６１１の処理内容を実行する。また、逐次実行の処理ループにある場合、命令実行後にポーリング処理Ｓ１６１２を実行する。これはsync命令語実行後のスレブＩＰＲの処理終了を確認する場合及び他のＩＰＲからの要求に｛ビジー｝応答する場合に、それぞれ通信ポートからの待ち行列を検査する必要から行なわれる。

【００９３】以上がＩＰＲの動作概要である。

【００９４】３－２．中間言語インタープリタの動作状態

本実施例のＩＰＲには初期起動という動作状態が有る。この動作状態は、中間言語オブジェクト１１０がオペレーティグシステム２０３の管理下で実行された直後にオペレーティグシステム２０３により作り出される動作状態である。オペレーティングシステム２０３は、中間言語オブジェクト１１０のファイル属性からこれがＩＰＲ１１１によって実行管理されるファイルであるという判断をする。オペレーティングシステム２０３は、この判断の結果、ＩＰＲ１１１を起動する。（この方法と別に、コマンドシェルと呼ばれるオペレーティングシステム用命令の通訳系を持つシステムではコマンドシェル用のテキストとして類似の操作を記述できる）。

【００９５】一方、ＩＰＲはその起動時にオペレーティングシステムから渡される引数を取得できる。上記操作によりこの引数として中間言語オブジェクトファイル１１０のファイル参照番号（ファイル管理情報）が渡される。このときＩＰＲは中間言語オブジェクト１１０をその作業領域にロードし実行する。ここで起動されたＩＰＲ１１１が以下マスタｉＰＲとなり、スレブｉＰＲであるＩＰＲ１１２に対し必要に応じてコマンドを送り処理の一部を分担させる。

【００９６】初期起動の状態にあるＩＰＲは状態コードが｛ビジー｝である。従って、他のＩＰＲが処理の並列実行の処理分担を要求し、｛コネクト・ロック｝メッセージをこのＩＰＲに発行しても要求が受けつけられない。これに対し初期起動の以外の動作状態であるＩＰＲは常にスレブＩＰＲとして動作することができる。

【００９７】３－３．並列記述部分の実行時処理

次に図１８を用いて並列実行時のＩＰＲの動作を説明する。

【００９８】マスタＩＰＲの命令語解析処理１８０１が中間言語オブジェクト１１０の記述中に命令語｛CALL addr(coexec)｝を検出した場合，cobegin文処理１８０２が呼び出される。cobegin文処理１８０２はシステムバス２０２を用いて全てのプロセッサに｛コネクト・ロック｝コマンドを送る。応答が｛ビジー｝であるか，または一定時間を経ても応答がない場合は，それらプロセッサが並列実行を分担できない状態に有ることが判断できる。一方で，プロセッサ資源の余裕度によっては複数のＩＰＲを実行中のプロセッサが存在し，複数の｛アクセプト｝メッセージを受信できる場合が有る。そこでマスタＩＰＲはその作業領域に実行時プロセス管理テーブル１７０１を作成する。実行時プロセス管理テーブル１７０１の内容は，｛アクセプト｝メッセージを受け取ることのできたプロセッサ番号及びプロセス番号と，このプロセスをスレブＩＰＲと見なして分散した（あるいはする予定の）処理ブロックのブロック番号と，各プロセスに対する要求待ち行列１８０７へのポインタである。

【００９９】並列実行の対象となるブロックの数が，獲得したスレブＩＰＲのプロセスの数より小さい場合，マスタＩＰＲは不必要なスレブＩＰＲに｛リリース｝メッセージを送りコネクトを開放する。これとは反対に並列実行すべき処理ブロック数が，獲得したスレブＩＰＲより多い場合は，獲得したスレブＩＰＲに対し待ち行列１８０７を作り，この待ち行列の要素に並列実行したい処理ブロックを追加する。この処理はcobegin文処理１８０２が呼び出したスケジューラ１８０３によって処理される。本実施例のスケジュールアルゴリズムは，スケジューラが検出した新たな要求は，最も短い長さの待ち行列１８０７に配置するという単純なアルゴリズムである。

【０１００】一方処理ブロックの一部はマスタＩＰＲにおいても処理可能である。このためスケジューラは述語展開処理１８０６を実行する。述語展開処理１８０６は，中間言語記述の

LOON foo
CALL addr(coexec)

という述語の並びを取り出し，

NOP
CALL addr(foo)

という述語の並びに置き換える。ここでfooはある手続きまたは関数の相対番地であり，命令語NOPは｛作用しない｝命令語を表す。述語展開処理１８０６により書き換えられた命令語記述は，再び命令語解析処理１８０１に返され，そこで評価される。このため，命令語解析処理１８０１および述語展開処理１８０６は，中間言語の命令ポインタ１７０８を処理内容に従い書き換える。

【０１０１】以上の処理において，並列実行の対象となる複数の処理ブロックをどの様な順序でマスタＩＰＲとスレブＩＰＲに分配するかはスケジューラ１８０３で決定される。本実施例のスケジューラ１８０３は後述する特殊な命令を使用しないかぎり，少なくとも一つの処理ブロックはマスタＩＰＲへの割り当てを行なう。

【０１０２】マスタＩＰＲは，スレブＩＰＲ１８０８に対する待ち行列１８０７に前述した｛ロード｝メッセージのデータ構造に従うデータ列と，｛実行｝メッセージをエンキューする。この処理要求を実行するスレブＩＰＲは処理終了後データ構造１５１０からなるメッセージを返す。この時の状態コードは正常終了であれば｛ダーン｝が返される。これに対し，何らかの実行時エラーを検出した場合は状態コード｛フェイル｝が返される。エラー処理に関して目的プログラムのソースコードが何らかの対応をする必要が有るが，これは従来のプログラミングと同じである。状態コード｛ダーン｝によって正常終了した処理は，もし処理ブロックが関数であれば関数の値をデータ１５１２によって返す。

【０１０３】並列実行を開始したマスタＩＰＲは，coend文によって並列実行の終了を待つ。既に説明した様にcoend文に対し生成される中間言語記述は｛CALL addr(sync)｝である。命令語解析処理１８０１は，この記述を検出するとsync文処理１８０５を呼び出す。sync文処理１８０５はスレブＩＰＲからの｛ダーン｝メッセージを待ち，実行時プロセッサ管理テーブル１７０１のプロセッサ番号，プロセス番号との一致を確認し処理終了を記録する。全ての処理ブロックの処理終了が確認されるとsync文処理１８０５は終了する。既に述べた様にマスタＩＰＲは，スケジューラ１８０３の制御に従い，少なくとも一つの処理ブロックを実行するのでＩＰＲ自体が逐次処理系であることからsync文処理１８０５が実行されるのはマスタＩＰＲの分担した処理ブロックの実行が終了した後である。

【０１０４】３－４．共有資源アクセスの実行時処理

次に，図１９の流れ図を用いてモニタ型手続き呼び出しによって共有変数へのアクセスが発生した場合の処理について説明する。命令語解析処理１８０１は，手続きappendまたはfetchの呼び出しを検出すると，ＩＰＲスタックの最上段からモニタ型の資源番号を取り出す（Ｓ１９０１）。この資源番号１００２をキーとして共有資源管理テーブル１００１を検索する（Ｓ１９０２）。検索結果として，その資源が宣言されたブロック番号１００４が取得できる（Ｓ１９０３）。このブロック番号を自ブロックの番号と比較し（Ｓ１９０４），一致する場合は通常の手続き呼び出しと同じ処理を行なう（Ｓ１９０７），それ以外であれば，ＩＰＲは自プロセスが実行中のブロック番号から実行時ブロック管理テーブル５０１の要素５０２にアクセスし，下位ブロックへの連鎖アドレス５０４からの連鎖５０８をたどる（Ｓ１９０５）。

この処理によってブロック番号の一致するブロックが見つかれば（Ｓ１９０６）手続き呼び出し処理を行なう（Ｓ１９０７）。連鎖５０８の終了まで検査しても一致するブロック番号を検出できない時は、実行時ブロック管理表によってモニタ型資源を保持するプロセッサ番号、プロセス番号を取り出し、この２つの番号で指定されるＩＰＲにメッセージ｛フェッチ｝あるいは｛アペンド｝を送り（Ｓ１９０８）処理結果のデータ列を受け取るまで待機する（Ｓ１９０９）。

【０１０５】一方、このメッセージを受け取ったＩＰＲ側では図１６に示す様にフェッチ処理Ｓ１６１３またはアペンド処理Ｓ１６１４を行なう。この処理はメッセージのデータ部分からモニタ型共有資源の資源番号を取り出し、この資源番号に従い共有資源管理テーブル１００１の登録を検査する。次に該当する要素を取り出し、そのデータ構造１００５からモニタ型手続きの実行コード上での番地計算を行ない、実際の処理を呼び出し、処理結果を状態コード｛ダーン｝と共に、要求側ＩＰＲに返す。

【０１０６】ここで、他のプロセッサにあるモニタ型変数にアクセスを行なった場合は、メッセージを受信し応答を返すＩＰＲの処理が逐次処理であるため、完全に排他制御される。また、他のプロセッサのＩＰＲから応答が有るまで要求側のＩＰＲは待機するため、要求側での追い越し処理の発生も防ぐことができる。同一プロセッサ上で複数のプロセスが生成され、複数のＩＰＲを実行中の場合もモニタ型では共有資源に対するアクセスを専用の処理に限定しており、この処理の実行がＩＰＲによって逐次的に行なわれるため排他制御できる。

【０１０７】一方、共有資源の中に実際のデータ列が書き込まれないうちに値を読み取ろうとしても処理内容は無効なものとなる。また、バッファが全て満たされた状態に更に書き込みを行なうと、先行するデータを（多重書き込みにより）失ってしまう。これらの不都合を防ぐには同期機構が必要であるが、モニタ型内部の手続きとして記述した待ち行列に対するwait及びsignal処理によって同期は維持される。

【０１０８】以上の処理によって本実施例では、排他制御を行ないながら複数個のＩＰＲの実行が可能となる。

【０１０９】３－５．実行プロセッサを明示する手続き

以上に説明した処理系は実際にリモートプロセッサに配置される処理ブロックも、またローカルプロセッサで実行を継続する処理ブロックも、中間言語記述の上では同様な命令語を使用する処理系であった。このため、中間言語記述の中でプロセッサを特定する記述や、処理ブロックをリモートプロセッサの局所メモリに転送するための記述を明示的に書く必要が無い処理系の実現方法を説明した。なぜなら、本発明がハードウェア変更により書き換えの必要の無い処理系の実現を目的としたからである。

【０１１０】一方、上記とは逆に専用プロセッサを付加した場合（特に信号処理、画像処理等の特定分野では重要であるが）、処理分散するプロセッサを明示的に特定したい場合がある。このために本実施例の処理系は組み込み関数として

```
function processor(slot:integer) : boolean;
```

を用意した。これは、システムバスの拡張基板用のスロットの番号を引数slot（整数型）で指定すると、その拡張基板用のスロットにプロセッサ基板が実装され、中間言語インタープリタが実行されているとき、論理［真］を返す関数である。この関数が呼び出されると、マスタＩＰＲは、プロセッサ番号を指定して｛コネクト・ロック｝メッセージを送る。この応答として｛アクセプト｝メッセージの応答があれば、関数の戻り値は｛TRUE｝が代入される。また、一定時間以上応答が無い場合、または応答が｛ビジー｝である場合は関数の戻り値は｛FALSE｝である。

【０１１１】さらに本実施例の処理系は、組み込み手続きとして

```
procedure distribute(slot:integer; procedure foo);
```

を用意した。これは拡張基板用のスロットの番号を引数slot（整数型）で指定し、かつそのスロットのハードウェア上のプロセッサに手続き引数fooで指定される手続きを配置し実行する手続きである。第２引数はfunction foo:tt;等の様に指定しても構わない。ここでfooは関数名、ttはデータ型名である。この手続きの実現のため、マスタＩＰＲはプロセッサ番号を指定して｛コネクト・ロック｝メッセージを送り、この応答として｛アクセプト｝メッセージの応答があれば続けて｛ロード｝及び｛実行｝メッセージを送る。この処理によって上記手続き中に手続き引数あるいは関数引数で指定された処理ブロックの実行を特定のハードウェアに割り当て処理する。上記の２つの組み込み処理を用い、プロセッサを特定した実行を記述することができる。次はその記述の一例である。引数ａ、ｂを持つ手続きPROC1と、引数ａを持つ手続きPROC2を並列実行する場合であり、システムバスのスロット４のプロセッサにPROC1を配置したいとすると次のように書ける。

【０１１２】

```
cobegin
if processor(4) then distribute(4, PROC1(a, b))
else PROC1(a, b);
PROC2(a);
coend;
```

この例では、拡張スロット４にプロセッサ基板が無い場合は手続きＰＲＯＣ１、PROC2は通常のシーケンスで並列実行される。

【０１１３】４．説明の補足

４－１．コンパイラ、インタープリタの使用する表

図２０を用いて本実施例の説明の中で示した各種表構造

について、その役割をまとめる。

【0114】ブロック番号管理テーブル801はコンパイラ102の作業領域に生成され、コンパイル処理が終了した時点で廃棄される。この表にはコンパイラ102の構文解析手段104において、ブロック番号、ブロック内変数表のポインタ等が記録される。またこの表構造を参照して実行時ブロック管理テーブル501のブロック間の連鎖生成が行なわれる。

【0115】実行時ブロック管理テーブル501は、コンパイラ102の作業領域に生成された後、オブジェクトファイル作成手段109によって中間言語オブジェクト110のファイルに出力される。さらに、中間言語オブジェクト110が中間言語インタープリタにロードされた後、中間言語インタープリタ内でも参照される。この表にはコンパイラ102の構文解析手段104によりブロック呼び出しの際の引数リストの解析結果、ブロック間の従属関係が記録される。

【0116】共有資源管理テーブル1001はコンパイラ102の作業領域に作成された後、オブジェクトファイル作成手段109によって中間言語オブジェクト110のファイルに出力される。この表は中間言語オブジェクト110が中間言語インタープリタにロードされた後、中間言語インタープリタ内でも参照される。中間言語インタープリタ中ではモニタ型手続き処理にあたり参照される。

【0117】実行時プロセッサ管理テーブル1701は、インタープリタ作業領域に実行時に生成される。この表は並列実行にあたり使用可能な（｛コネクト・ロック｝できた）プロセッサの管理および、並列実行した結果のスケジュール管理に使用される。

【0118】上記表構造に加え、本実施例ではコンパイラ102において変数名、ブロック名等の識別子の管理に名前表およびブロック内変数表を用いた。

【0119】4－2．本実施例の展開

4－2－1．排他制御と共有資源管理

上記の説明において、共有資源への排他制御、同期機構としてモニタ型を挙げて説明した。これは本実施例において実現容易なものとして使用した。しかし、本実施例と同一の構成において直ちに周知のセマフォに対しても適用できる。このためには、どのブロックの管理下に有るセマフォにアクセスするかを本実施例の実行時ブロック管理テーブル501と同様のデータ構造で記録し、自プロセッサ外のセマフォに関しては通信を伴ってリモートにアクセスする組み込み手続きを用意するだけで良い。

【0120】周知の排他制御、同期機構はセマフォを用いて置き換え可能な場合が多い。このことから本実施例はモニタ型以外の共有変数実現方法にも応用可能であると言うことができる。

【0121】また本実施例で示したモニタ型において、

バッファメモリ割り当てを取り去った構造を用いた場合、あるデータ型に結び付けられた特定手続きによる同期的な通信経路が中間言語インタープリタにより実現されたと見なすことができる。従って本発明の中間言語インタープリタは、インタープリタとインタープリタの間でFIFO等のメモリ構造を用い通信経路を実現した場合に対しても実装することができる。この場合、共有変数を持たず、通信によって並列実行手続き間のデータ授受をおこなう仕様のプログラム言語のコンパイラであっても、本実施例と同様な中間言語出力を行なうことで複数のプロセッサに実装した中間言語インタープリタにおいて実行可能であると言える。

【0122】4－2－2．中間言語インタープリタの実装

上記実施例の示したものと等価な中間言語インタープリタは、複数プロセッサからなる処理系に実現されても良い。例えば4つのプロセッサからなるコンピュータ上に並列処理を実現するカーネルプログラムを実装し、このカーネルプログラム上に中間言語インタープリタプログラムを走らせることが考えられる。これを更に発展させると、トポロジーの異なる別々のマルチプロセッサシステム上にそれぞれ本発明の中間言語インタープリタを実装し、トポロジー混在環境で実行可能な並列プログラムを記述する事ができる。

【0123】この方法を使用すれば、画像処理の高速処理に有利なシストリックアレイプロセッサシステムと、信号処理、高速フーリエ変換処理などの処理に有利なベクトルプロセッサシステムを結合したシステム上で記述可能な高級言語を実現することができる。

【0124】従来のコンパイラの中で、例えばベクトルプロセッサの使用を前提として開発されたコンパイラの出力コードは、ベクトルプロセッサ上で実行される機械語コードを発生する。このオブジェクトコードはターゲットシステムに依存したコードである。この種のコンパイラの出力結果は、ハードウェアの仕様の変更に対しては再コンパイルしないと最適コードとならない。例えば、スカラープロセッサのみのハードウェアを実行環境としてコンパイルされたオブジェクトコードを適用していたコンピュータに新たにベクトルプロセッサを付加したとしても、ベクトルプロセッサをサポートするコンパイラによって再度コンパイルし、オブジェクトコードを生成し直さないと実行速度の向上効果は得られない。

【0125】これに対し、本実施例で示したコンパイラは中間コードインタプリタにより実行されるコードを発生する。また中間コードは実行時にプロセッサに動的にスケジュールされる。この2つの特徴からコンパイル済みのオブジェクトコードを適用しているシステムに対し、中間コードインタープリタを実装したハードウェア（回路基板等）を付加した場合、再度コンパイルを行ない目的プログラムのオブジェクトコードを生成し直す手

順が不要である。中間コードインタープリタを実装したハードウェアが付加された時点から目的プログラムのオブジェクトコードは、付加ハードウェアのプロセッサと従来から運用していたプロセッサによるマルチプロセッサ系によって実行される。このことは、商用アプリケーションを販売し運用する段階で極めて有利である。なぜなら、商用アプリケーションのほとんどは、プログラム著作権を保護するためソースコードを含めないオブジェクトコードのみの状態で使用者に販売されるためである。本実施例のコンパイラによって開発されたオブジェクトコードであれば、使用者がハードウェア付加を行なってもアプリケーションソフトウェアのオブジェクトコードを変更する必要が無い。これにより、保守の容易さ、運用の簡便さの両面で優れるという効果が生じる。

【０１２６】

【発明の効果】本発明は、並列実行時に共有される変数へのアクセスを排他制御する機構と、中間言語の命令語によって記述されたオブジェクトコードを生成するコンパイラと、前記コンパイラの出力した中間言語を実行するインタープリタと、前記インタープリタにより実現される並列実行の処理ブロック割り当てを動的に行なう機構とによって構成されているため、並列処理系のハードウェアの構成が変わった場合でもこのコンパイラを用いて開発されたオブジェクトコードを再コンパイルする事無く並列処理を実行することが可能である。

【０１２７】言い替えるなら、使用者が処理速度改善のためアクセラレータ等の付加ハードウェアを追加した場合でも、本発明による処理系を実装したパーソナルコンピュータ向けのアプリケーションプログラムについては、ハードウェア追加によるプログラムの変更を必要とせずに追加したハードウェアによる処理速度の改善を享受できる。

【図面の簡単な説明】

【図１】本発明の実施例として好適なマルチプロセッサ処理装置と、そのコンパイラ処理系の構成図。

【図２】図１のコンパイラ処理系で開発したオブジェクトコードを実行するマルチプロセッサ処理装置であるパーソナルコンピュータの構成図。

【図３】モニタ型の説明図。

【図４】構文解析手段１０４の処理の流れ図。

【図５】実行時ブロック管理テーブルの説明図。

【図６】例として挙げたプログラムの説明図。

【図７】例として挙げたプログラムのブロックの従属関係の説明図。

【図８】ブロック管理テーブルの説明図。

【図９】ブロック間の従属関係を取り出す処理の流れ図。

【図１０】共有資源管理テーブルの説明図。

【図１１】中間言語の機能の説明図。

【図１２】中間言語発生の処理の説明図。

【図１３】中間言語発生手順の説明図。

【図１４】中間言語処理時のスタック状態の説明図。

【図１５】中間言語インタープリタを形成するプロセス間の通信に使用するデータ構造の説明図。

【図１６】中間言語インタープリタの処理の流れ図。

【図１７】中間言語インタープリタを実現するプロセスの構成図。

【図１８】cobegin文、sync文処理の説明図。

【図１９】共有資源アクセス処理の実行時の流れ図。

【図２０】表構造の説明図。

【符号の説明】

１０１…ソースコード
１０２…コンパイラ
１０３…字句解析手段
１０４…構文解析手段
１０５…共有メモリ資源検出手段
１０６…並列実行検出手段
１０７…コード生成手段
１０８…コード生成手段
１０９…オブジェクトファイル作成手段
１１０…中間言語オブジェクト
１１１…中間言語インタープリタ
１１２…中間言語インタープリタ
１１３…プロセッサ
１１５…通信経路
２００…パーソナルコンピュータ
２０１…付加ハードウェア
２０２…システムバス
２０３…オペレーティングシステム
３００…説明のためのソースコード
５０１…実行時ブロック管理テーブル
５０３…ブロック番号
５０４…下位ブロックへの追跡アドレス
５０７…引数リストへの追跡アドレス
５０８…連鎖
５０９…連鎖
８０１…ブロック番号管理テーブル
８０２…ブロック番号
８０３…ブロックレベル
８０４…テーブルエントリ
１００１…共有資源管理テーブル
１００２…資源番号
１００３…共有変数名
１００４…共有資源が宣言されたブロック番号
１００６…モニタ型手続きの実行コード
１２０３…引数リスト
１４０１…スタック使用の履歴
１５０１…データ構造
１５０２…コマンド
１５０３…プロセッサ番号

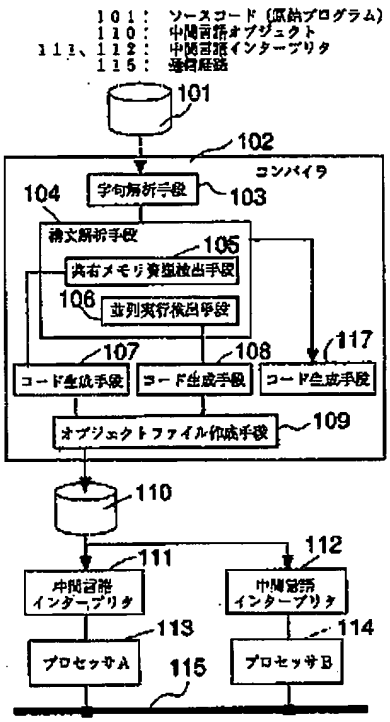
27

１５０４…プロセス番号
１５０６…データ長
１５０７…オブジェクトコード
１５０８…スタック初期化データ
１５０９…テーブル初期化データ
１５１１…状態コード
１７０１…実行時プロセッサ管理テーブル
１７０２…プロセス

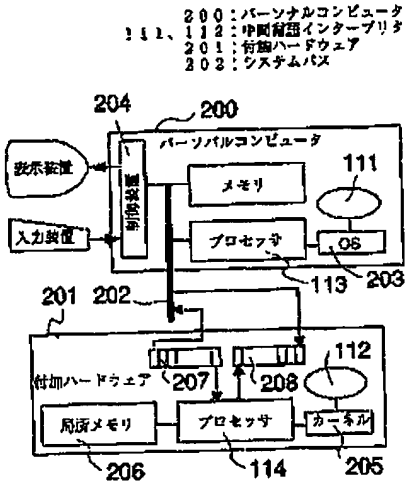
28

＊１７０６…コード領域
１７０７…スタック領域
１７０８…命令ポインタ
１７０９…スタックポインタ
１８０１…命令語解析処理
１８０３…スケジューラ
１８０４…述語展開処理
＊　１８０８…他の中間言語インタープリタ

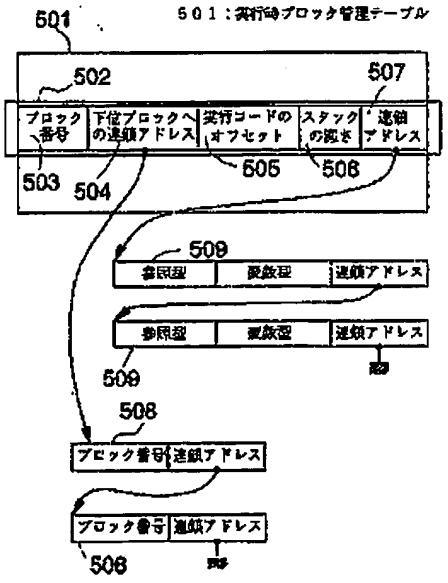
【図１】



【図２】

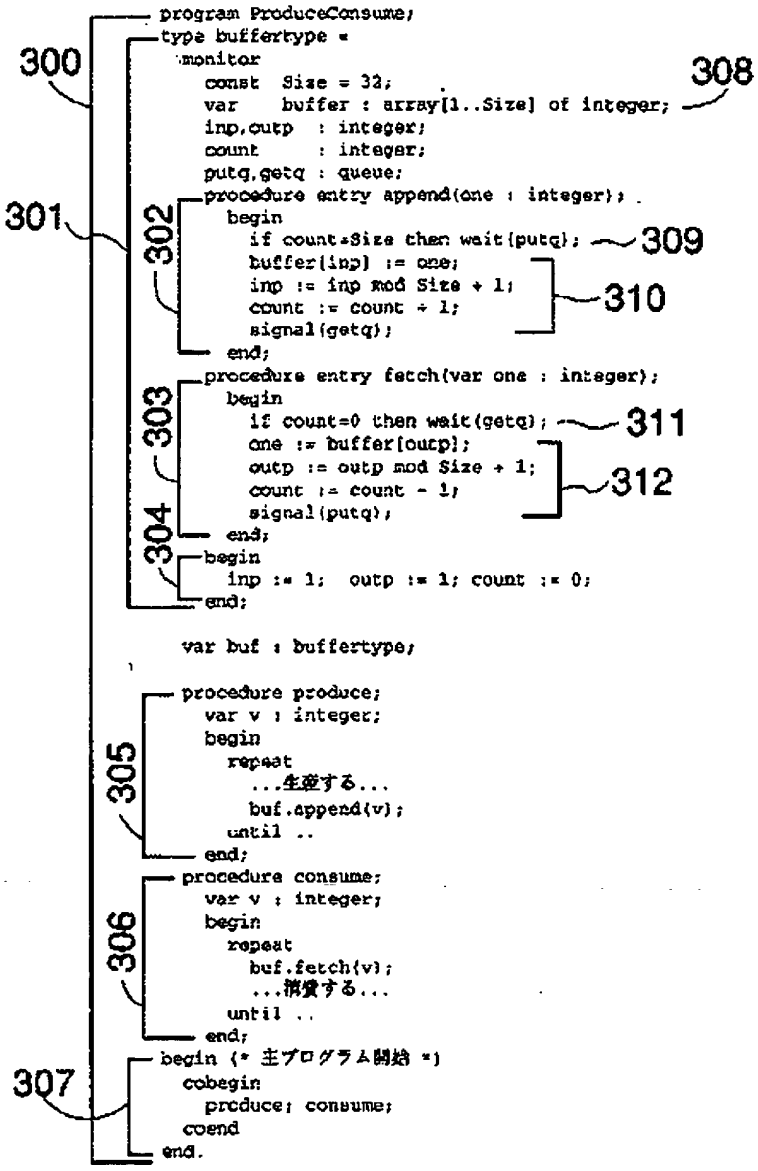


【図５】

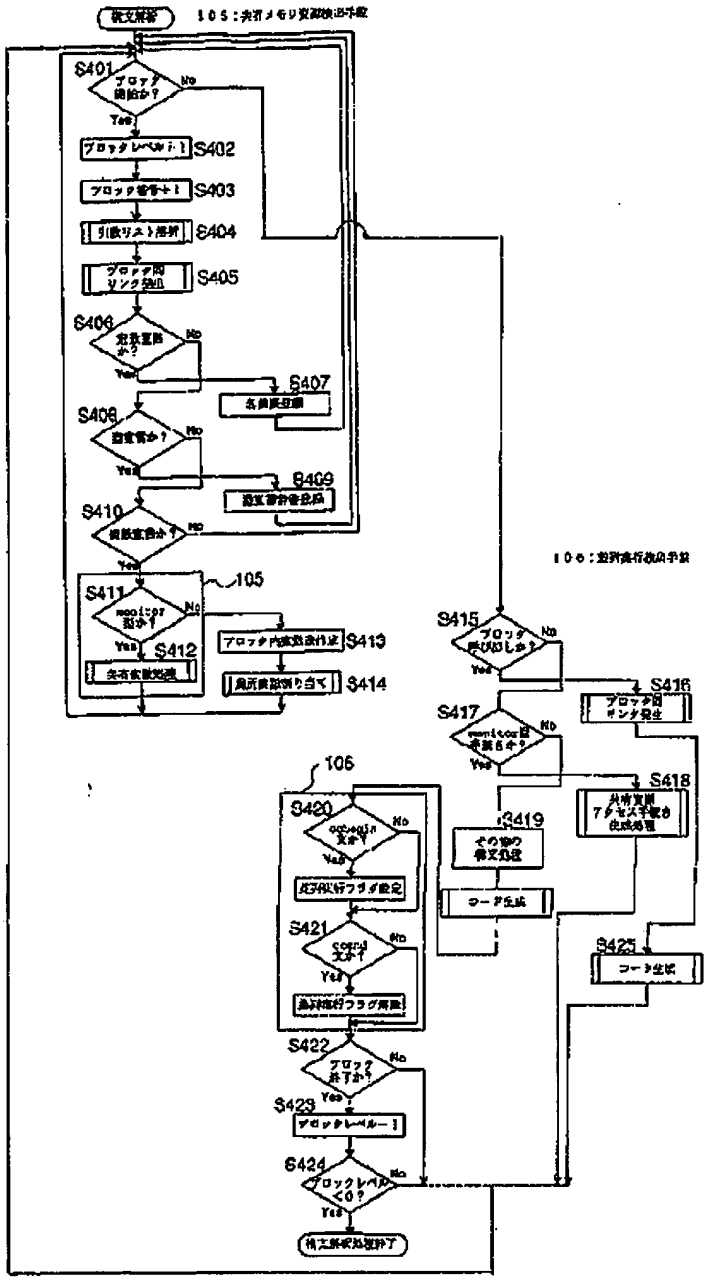


【図3】

３０１：モニタ型の型宣言



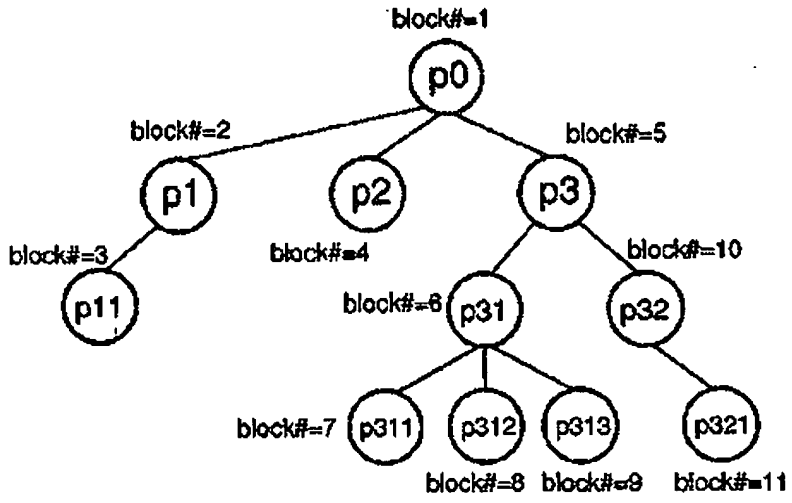
【図4】


S401
S402
S403
S404
S405
S406
S407
S408
S409
S410
S411
S412
S413
S414
105
S415
S416
S417
S418
S419
106
S420
S421
S422
S423
S424
S425

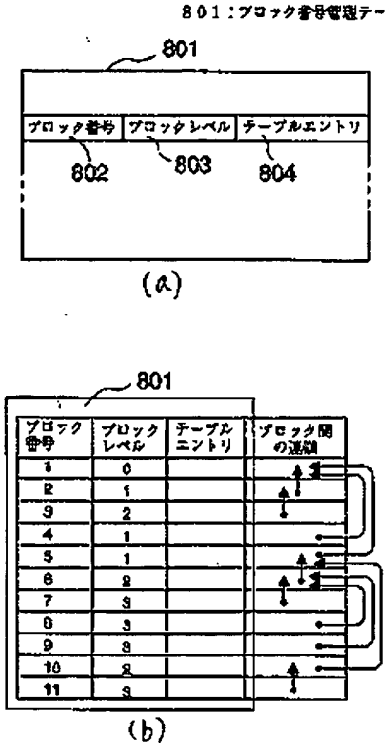
【図6】

```
program p0;

  procedure p1;
    function p11: integer;
      begin ..... end;
   begin
     ....;
   end;

  procedure p2;
   begin ..p1;... end;

  procedure p3;
    procedure p31;
        procedure p311;
             begin .... end;
        procedure p312;
             begin .... end;
        function p313: integer;
             begin .... end;
     begin ... end;
    procedure p32;
        procedure p321;
             begin .... end;
     begin ... end;
   begin
     ....
   end;

begin
  .... p3; .....;
end.
```

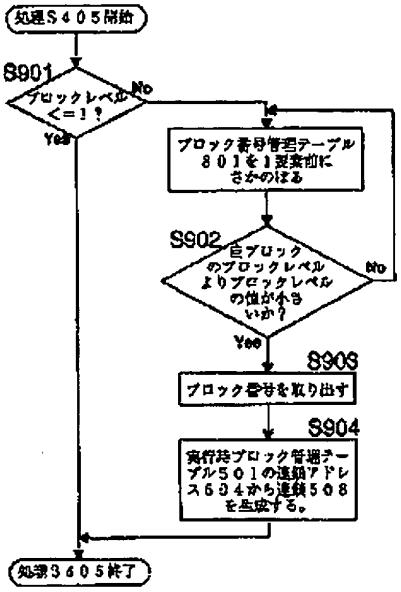
【図7】



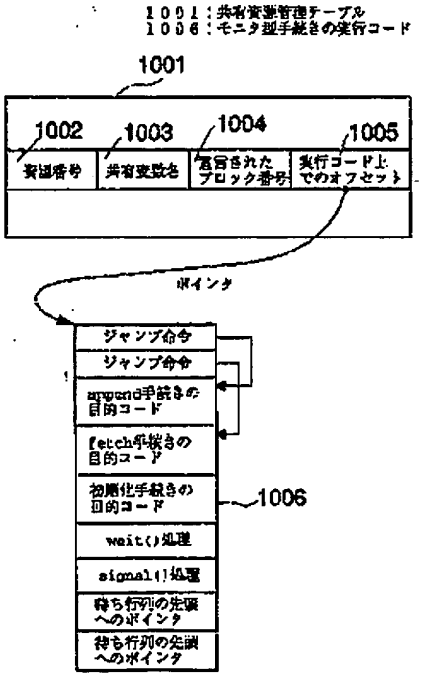
【図８】


８０１：ブロック番号管理テーブル
801
ブロック番号
ブロックレベル
テーブルエントリ
802
803
804
(a)
801
ブロック番号
ブロックレベル
テーブルエントリ
ブロック間の連鎖
(b)

【図９】


処理S４０５開始
S901
ブロックレベル<=1？
No
Yes
ブロック番号管理テーブル８０１を1要素前にさかのぼる
S902
自ブロックのブロックレベルよりブロックレベルの値が小さいか？
No
Yes
S903
ブロック番号を取り出す
S904
実行時ブロック管理テーブル５０１の連鎖アドレス５０４から連鎖５０８を生成する。
処理３６０５終了

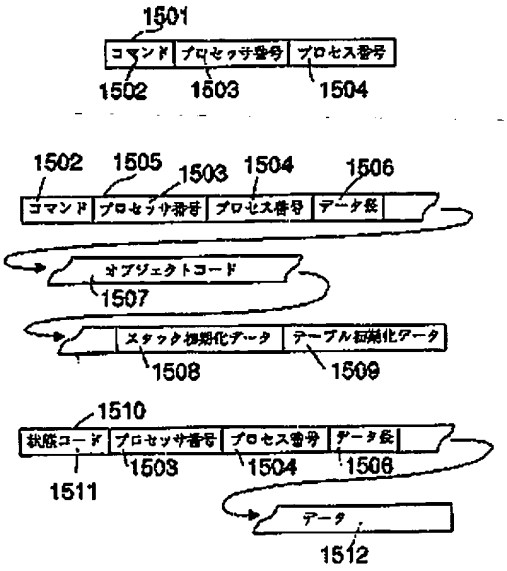
【図１０】


１００１：共有資源管理テーブル
１００６：モニタ型手続きの実行コード
1001
1002
1003
1004
1005
資源番号
共有変数名
割当てられたブロック番号
実行コード上でのオフセット
ポインタ
ジャンプ命令
ジャンプ命令
signal手続きの目的コード
fetch手続きの目的コード
初期化手続きの目的コード
1006
wait()処理
signal()処理
待ち行列の先頭へのポインタ
待ち行列の先端へのポインタ

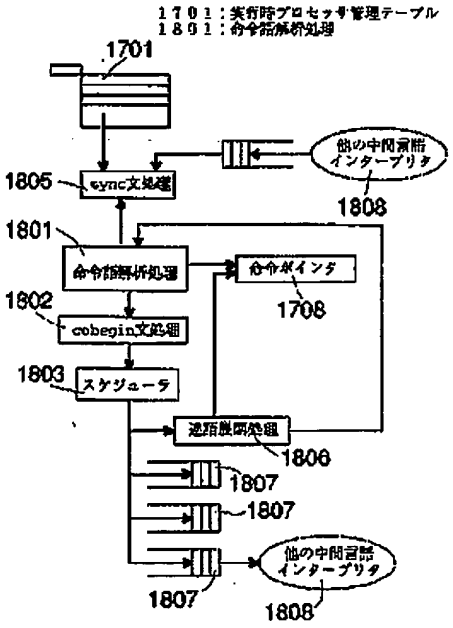
【図11】

命令語	名前の意味	動作の概要
LODV	Load Value	(TOP-1) ← value SP ← SP-1
LODA	Load Address	(TOP-1) ← addr(x) SP ← SP-1
LODN	Load Number of an object	find number of an obj. (TOP-1) ← number of an obj
LOD n	Load object	(TOP-1) ← (TOP+n) SP ← SP-1
ALOC n	Alocate	SP ← SP-n
UALC n	UnAlocate	SP ← SP+n
CALL	Call subroutine	ar[1] ← PC+1 PC ← addr(x)
RET	return from ..	PC ← ar[1]
ADD	Addition	(TOP) ← (TOP+1)+(TOP)
RELS	Release	release connection

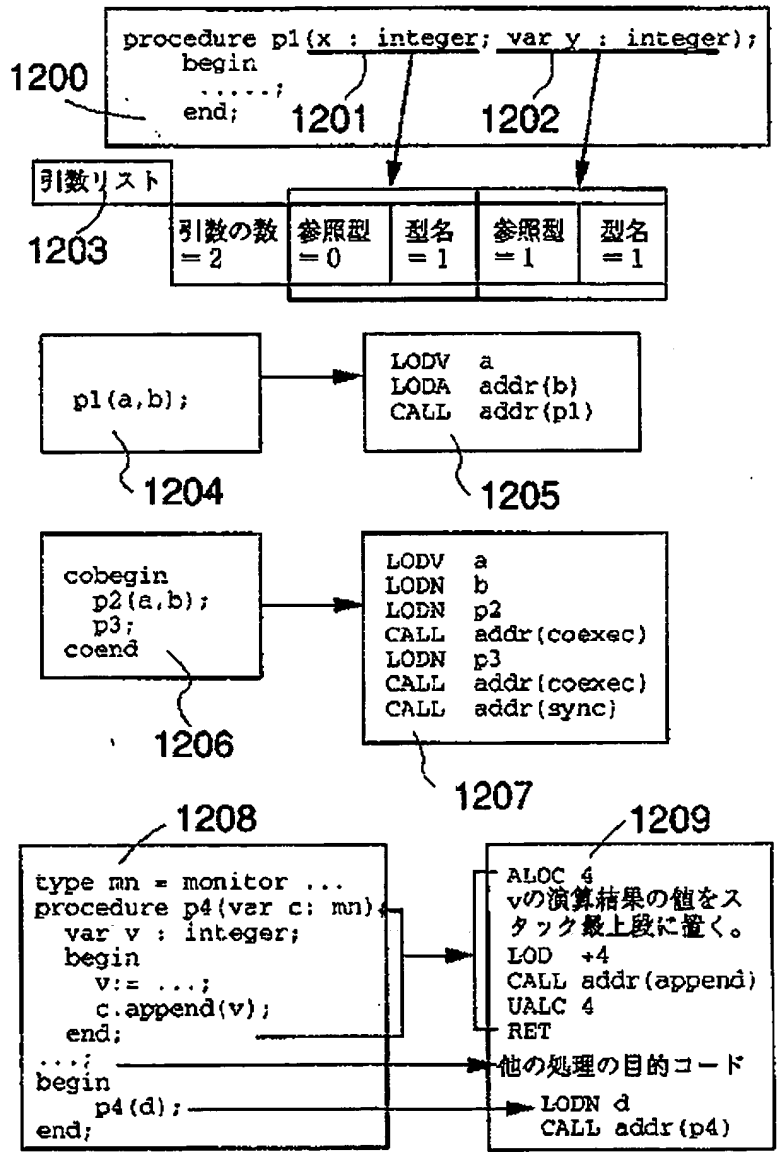
【図15】



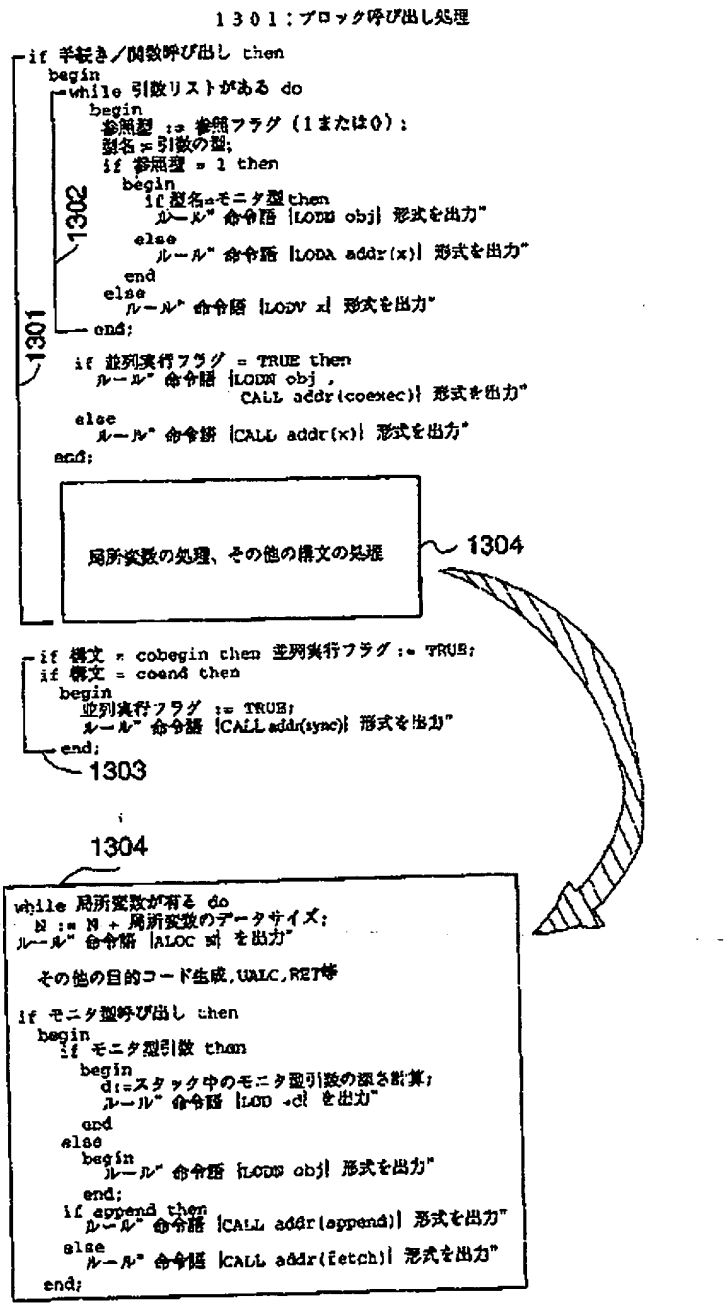
【図18】



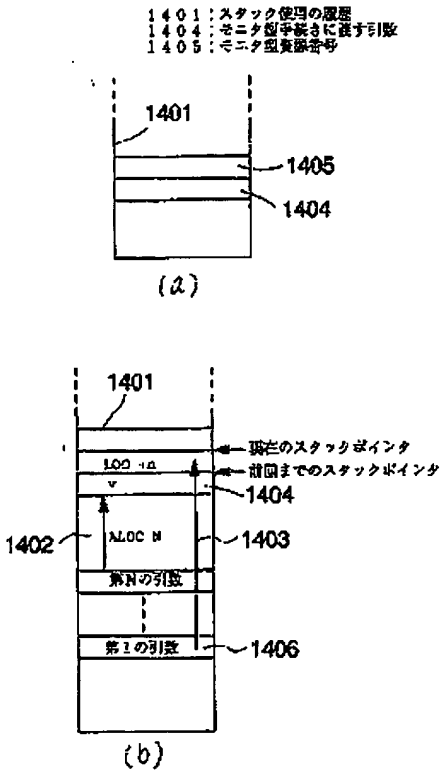
[図12]


procedure p1(x : integer; var y : integer);
begin
.....;
end;
1200
1201
1202
引数リスト
1203
引数の数 ＝2
参照型 ＝0
型名 ＝1
参照型 ＝1
型名 ＝1
p1(a,b);
1204
LODV a
LODA addr(b)
CALL addr(p1)
1205
cobegin
p2(a,b);
p3;
coend
1206
LODV a
LODN b
LODN p2
CALL addr(coexec)
LODN p3
CALL addr(coexec)
CALL addr(sync)
1207
1208
type mn = monitor ...
procedure p4(var c: mn);
var v : integer;
begin
v:= ...;
c.append(v);
end;
...;
begin
p4(d);
end;
1209
ALOC 4
vの演算結果の値をスタック最上段に置く。
LOD +4
CALL addr(append)
UALC 4
RET
他の処理の目的コード
LODN d
CALL addr(p4)

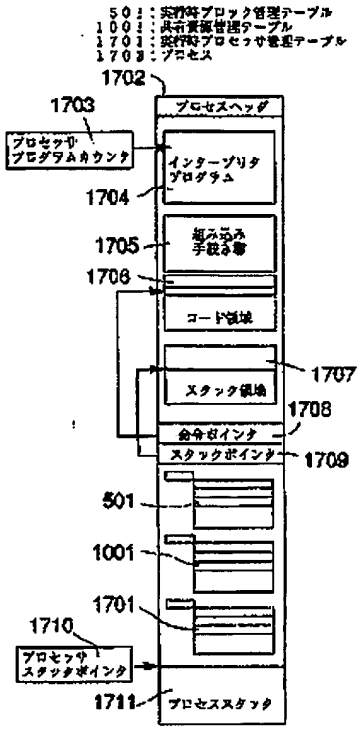
【図13】


1301：ブロック呼び出し処理
if 手続き／関数呼び出し then
begin
while 引数リストがある do
begin
参照型 := 参照フラグ（1または0）;
型名 := 引数の型;
if 参照型 = 1 then
begin
if 型名=モニタ型 then
ルール" 命令語 |LODM obj| 形式を出力"
else
ルール" 命令語 |LODA addr(x)| 形式を出力"
end
else
ルール" 命令語 |LODV x| 形式を出力"
end;
1301
1302
if 並列実行フラグ = TRUE then
ルール" 命令語 |LODM obj ,
CALL addr(coexec)| 形式を出力"
else
ルール" 命令語 |CALL addr(x)| 形式を出力"
end;
局所変数の処理、その他の構文の処理
1304
if 構文 = cobegin then 並列実行フラグ := TRUE;
if 構文 = coend then
begin
並列実行フラグ := TRUE;
ルール" 命令語 |CALL addr(sync)| 形式を出力"
end;
1303
1304
while 局所変数が有る do
N := N + 局所変数のデータサイズ;
ルール" 命令語 |ALOC N| を出力"
その他の目的コード生成,UALC,RET等
if モニタ型呼び出し then
begin
if モニタ型引数 then
begin
di:=スタック中のモニタ型引数の深さ計算;
ルール" 命令語 |LOD di| を出力"
end
else
begin
ルール" 命令語 |LODM obj| 形式を出力"
end;
if append then
ルール" 命令語 |CALL addr(append)| 形式を出力"
else
ルール" 命令語 |CALL addr(fetch)| 形式を出力"
end;

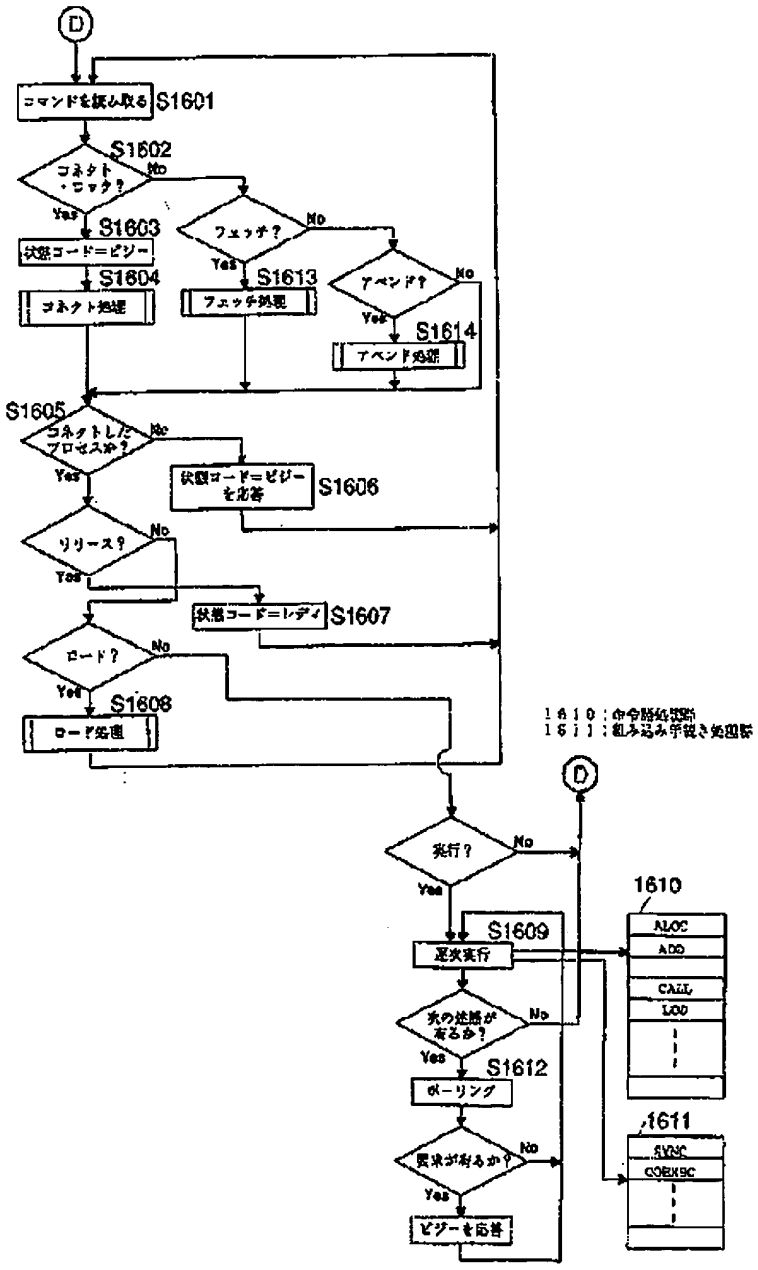
【図１４】


1401：スタック使用の履歴
1404：モニタ型手続きに渡す引数
1405：モニタ型資源番号
1401
1405
1404
(a)
1401
現在のスタックポインタ
前回までのスタックポインタ
1404
ALOC N
1402
1403
第Nの引数
第1の引数
1406
(b)

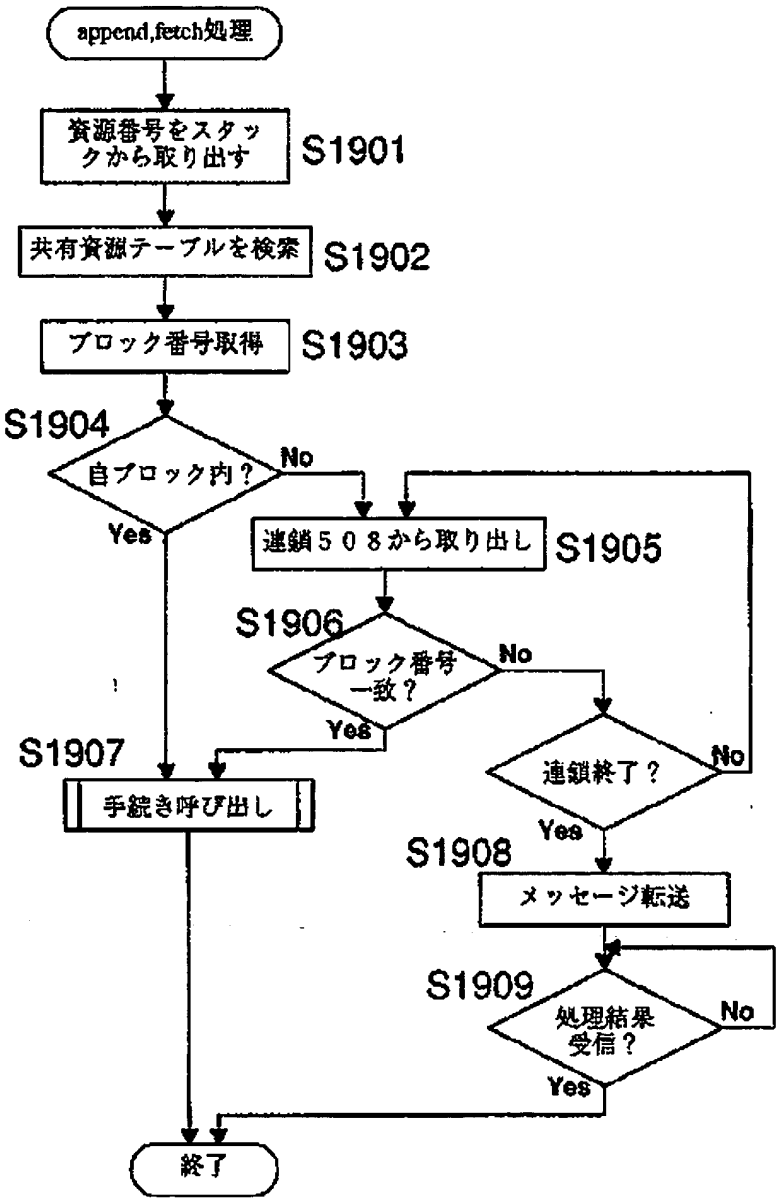
【図１７】


501：実行単ブロック管理テーブル
1001：共有資源管理テーブル
1701：実行単プロセッサ管理テーブル
1702：プロセス
1702
プロセスヘッダ
1703
プロセッサ
プログラムカウンタ
インタープリタ
プログラム
1704
1705
1706
コード領域
スタック領域
1707
1708
1709
スタックポインタ
501
1001
1701
1710
プロセッサ
スタックポインタ
1711
プロセススタック

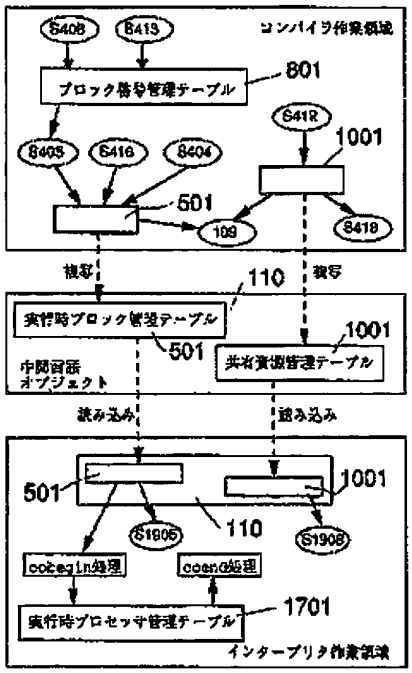
【図１６】


S1601
S1602
S1603
S1604
S1605
S1606
S1607
S1608
S1609
S1610
S1611
S1612
S1613
S1614
Yes
No
1610
1611
S1612

【図19】


append,fetch処理
資源番号をスタックから取り出す
S1901
共有資源テーブルを検索
S1902
ブロック番号取得
S1903
S1904
自ブロック内？
No
Yes
連鎖５０８から取り出し
S1905
S1906
ブロック番号一致？
No
Yes
S1907
手続き呼び出し
連鎖終了？
No
Yes
S1908
メッセージ転送
S1909
処理結果受信？
No
Yes
終了

【図２０】


S408
S413
コンパイラ作業領域
ブロック指令管理テーブル
801
S412
1001
S405
S416
S404
501
109
S418
複写
110
複写
実行時ブロック管理テーブル
501
1001
共有資源管理テーブル
中間言語
オブジェクト
読み込み
読み込み
501
1001
S1905
110
S1908
coExeIn処理
coend処理
実行時プロセッサ管理テーブル
1701
インタープリタ作業領域